Blurb Parser 1/blurb

Purpose

To read and follow the instructions in the blurb file, our main input.

§**1. Reading the file.** We divide the file into blurb commands at line breaks, so:

```
void parse_blurb_file(char *in) {
    file_read(in, "can't open blurb file", TRUE, interpret, 0);
    set_error_position(NULL);
}
```

The function parse_blurb_file is called from 1/main.

§**2.** The sequence of values enumerated here must correspond exactly to indexes into the syntaxes table below.

```
define author_COMMAND 0
define auxiliary_COMMAND 1
define base64_COMMAND 2
define copyright_COMMAND 3
define cover_COMMAND 4
define css_COMMAND 5
define ifiction_COMMAND 6
define ifiction_public_COMMAND 7
define ifiction_file_COMMAND 8
define interpreter_COMMAND 9
define palette_COMMAND 10
define palette_16_bit_COMMAND 11
define palette_32_bit_COMMAND 12
define picture_scaled_COMMAND 13
define picture_COMMAND 14
define placeholder_COMMAND 15
define project_folder_COMMAND 16
define release_COMMAND 17
define release_file_COMMAND 18
define release_file_from_COMMAND 19
define release_source_COMMAND 20
define release_to_COMMAND 21
define resolution_max_COMMAND 22
define resolution_min_max_COMMAND 23
define resolution_min_COMMAND 24
define resolution_COMMAND 25
define solution_COMMAND 26
define solution_public_COMMAND 27
define sound_music_COMMAND 28
define sound_repeat_COMMAND 29
define sound_forever_COMMAND 30
define sound_song_COMMAND 31
define sound_COMMAND 32
```

```
define source_COMMAND 33
define source_public_COMMAND 34
define status_COMMAND 35
define status_alternative_COMMAND 36
define status_instruction_COMMAND 37
define storyfile_include_COMMAND 38
define storyfile_COMMAND 39
define storyfile_leafname_COMMAND 40
define template_path_COMMAND 41
define website_COMMAND 42
```

§**3.** A single number specifying various possible combinations of operands:

```
define OPS_NO 1
define OPS_1TEXT 2
define OPS_2TEXT 3
define OPS_1NUMBER 4
define OPS_2NUMBER 5
define OPS_1NUMBER_1TEXT 6
define OPS_1NUMBER_2TEXTS 7
define OPS_1NUMBER_1TEXT_1NUMBER 8
define OPS_3NUMBER 9
define OPS_3TEXT 10
```

§**4.** Each legal command syntax is stored as one of these structures. We will be parsing commands using the C library function `sscanf`, which is a little idiosyncratic. It is, in particular, not easy to find out whether `sscanf` successfully matched the whole text, since it returns only the number of variable elements matched, so that it can't tell the difference between `do %n` and `do %n quickly`, say. The text "do 12" would match against both and return 1 in each case. To get around this, we end the prototype with a spurious `" %n"`. The space can match against arbitrary white space, including none at all, and `%n` is not strictly a match – instead it sets the number of characters from the original command which have been matched. It would be nice to use `sscanf`'s return value to test whether the `%n` has been reached, but this is unsafe because the `sscanf` specification is ambiguous as to whether or not a `%n` counts towards the return value; the `man` page openly admits that people aren't sure whether it does or doesn't. So we ignore the return value of `sscanf` as meaningless, and instead test the value set by `%n` to see if it's the length of the original text.

```
typedef struct blurb_command {
    char *explicated;                    plain English form of the command
    char *prototype;                     sscanf prototype
    int operands;                        one of the above OPS_* codes
    int deprecated;
} blurb_command;
```

The structure blurb_command is private to this section.

§**5.** And here they all are. They are tested in the sequence given, and the sequence must exactly match the numbering of the `*_COMMAND` values above, since those are indexes into this table.

In blurb syntax, a line whose first non-white-space character is an exclamation mark `!` is a comment, and is ignored. (This is the I6 comment character, too.) It appears in the table as a command but, as we shall see, has no effect.

```
blurb_command syntaxes[] = {
    { "author \"name\"", "author \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "auxiliary \"filename\" \"description\"",
            "auxiliary \"%[^\"]\" \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "base64 \"filename\" to \"filename\"",
            "base64 \"%[^\"]\" to \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "copyright \"message\"", "copyright \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "cover \"filename\"", "cover \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "css", "css %n", OPS_NO, FALSE },
    { "ifiction", "ifiction %n", OPS_NO, FALSE },
    { "ifiction public", "ifiction public %n", OPS_NO, FALSE },
    { "ifiction \"filename\" include", "ifiction \"%[^\"]\" include %n", OPS_1TEXT, FALSE },
    { "interpreter \"interpreter-name\" \"vm-letter\"",
        "interpreter \"%[^\"]\" \"%[gz]\" %n", OPS_2TEXT, FALSE },
    { "palette { details }", "palette {%[^}]} %n", OPS_1TEXT, TRUE },
    { "palette 16 bit", "palette 16 bit %n", OPS_NO, TRUE },
    { "palette 32 bit", "palette 32 bit %n", OPS_NO, TRUE },
    { "picture N \"filename\" scale ...",
            "picture %d \"%[^\"]\" scale %s %n", OPS_1NUMBER_2TEXTS, TRUE },
    { "picture N \"filename\"", "picture %d \"%[^\"]\" %n", OPS_1NUMBER_1TEXT, FALSE },
    { "placeholder [name] = \"text\"", "placeholder [%[A-Z]] = \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "project folder \"pathname\"", "project folder \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release \"text\"", "release \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release file \"filename\"", "release file \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release file \"filename\" from \"template\"",
            "release file \"%[^\"]\" from \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "release source \"filename\" using \"filename\" from \"template\"",
            "release source \"%[^\"]\" using \"%[^\"]\" from \"%[^\"]\" %n", OPS_3TEXT, FALSE },
    { "release to \"pathname\"", "release to \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "resolution NxN max NxN", "resolution %d max %d %n", OPS_2NUMBER, TRUE },
    { "resolution NxN min NxN max NxN", "resolution %d min %d max %d %n", OPS_3NUMBER, TRUE },
    { "resolution NxN min NxN", "resolution %d min %d %n", OPS_2NUMBER, TRUE },
    { "resolution NxN", "resolution %d %n", OPS_1NUMBER, TRUE },
    { "solution", "solution %n", OPS_NO, FALSE },
    { "solution public", "solution public %n", OPS_NO, FALSE },
    { "sound N \"filename\" music", "sound %d \"%[^\"]\" music %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\" repeat N",
            "sound %d \"%[^\"]\" repeat %d %n", OPS_1NUMBER_1TEXT_1NUMBER, TRUE },
    { "sound N \"filename\" repeat forever",
            "sound %d \"%[^\"]\" repeat forever %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\" song", "sound %d \"%[^\"]\" song %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\"", "sound %d \"%[^\"]\" %n", OPS_1NUMBER_1TEXT, FALSE },
    { "source", "source %n", OPS_NO, FALSE },
    { "source public", "source public %n", OPS_NO, FALSE },
    { "status \"template\" \"filename\"", "status \"%[^\"]\" \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "status alternative ||link to Inform documentation||",
        "status alternative ||%[^|]|| %n", OPS_1TEXT, FALSE },
```

```
    { "status instruction ||link to Inform source text||",
        "status instruction ||%[^|]|| %n", OPS_1TEXT, FALSE },
    { "storyfile \"filename\" include", "storyfile \"%[^\"]\" include %n", OPS_1TEXT, FALSE },
    { "storyfile \"filename\"", "storyfile \"%[^\"]\" %n", OPS_1TEXT, TRUE },
    { "storyfile leafname \"leafname\"", "storyfile leafname \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "template path \"folder\"", "template path \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "website \"template\"", "website \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { NULL, NULL, OPS_NO, FALSE }
};
```

§**6. Summary.** For the `-help` information:

```
void summarise_blurb(void) {
    int t;
    printf("\nThe blurbfile is a script of commands, one per line, in these forms:\n");
    for (t=0; syntaxes[t].prototype; t++)
        if (syntaxes[t].deprecated == FALSE)
            printf("  %s\n", syntaxes[t].explicated);
    printf("\nThe following syntaxes, though legal in Blorb 2001, are not supported:\n");
    for (t=0; syntaxes[t].prototype; t++)
        if (syntaxes[t].deprecated == TRUE)
            printf("  %s\n", syntaxes[t].explicated);
}
```

The function summarise_blurb is called from 1/main.

§**7. The interpreter.** The following routine is called for each line of the blurb file in sequence, including any blank lines.

```
void interpret(char *command, text_file_position *tf) {
    set_error_position(tf);
    if (command == NULL) fatal("null blurb line");
    command = trim_white_space(command);
    if (command[0] == 0) return;                          thus skip a line containing only blank space
    if (command[0] == '!') return;                                thus skip a comment line
    if (trace_mode) fprintf(stdout, "! %03d: %s\n", tfp_get_line_count(tf), command);
    int outcome = -1;                                     which of the legal command syntaxes is used
    char text1[MAX_TEXT_FILE_LINE_LENGTH], text2[MAX_TEXT_FILE_LINE_LENGTH],
        text3[MAX_TEXT_FILE_LINE_LENGTH];
    text1[0] = 0; text2[0] = 0; text3[0] = 0;
    int num1 = 0, num2 = 0, num3 = 0;
    ⟨Parse the command and set operands appropriately 8⟩;
    ⟨Take action on the command 9⟩;
}
```

The function interpret is.

§**8.** Here we set `outcome` to the index in the syntaxes table of the line matched, or leave it as -1 if no match can be made. Text and number operands are copied in `text1`, `num1`, ..., accordingly.

⟨Parse the command and set operands appropriately 8⟩ ≡

```
int t;
for (t=0; syntaxes[t].prototype; t++) {
    char *pr = syntaxes[t].prototype;
    int nm = -1;                                                    number of characters matched
    switch (syntaxes[t].operands) {
        case OPS_NO: sscanf(command, pr, &nm); break;
        case OPS_1TEXT: sscanf(command, pr, text1, &nm); break;
        case OPS_2TEXT: sscanf(command, pr, text1, text2, &nm); break;
        case OPS_1NUMBER: sscanf(command, pr, &num1, &nm); break;
        case OPS_2NUMBER: sscanf(command, pr, &num1, &num2, &nm); break;
        case OPS_1NUMBER_1TEXT: sscanf(command, pr, &num1, text1, &nm); break;
        case OPS_1NUMBER_2TEXTS: sscanf(command, pr, &num1, text1, text2, &nm); break;
        case OPS_1NUMBER_1TEXT_1NUMBER: sscanf(command, pr, &num1, text1, &num2, &nm); break;
        case OPS_3NUMBER: sscanf(command, pr, &num1, &num2, &num3, &nm); break;
        case OPS_3TEXT: sscanf(command, pr, text1, text2, text3, &nm); break;
        default: fatal("unknown operand type");
    }
    if (nm == strlen(command)) { outcome = t; break; }
}
if ((strlen(text1) >= MAX_FILENAME_LENGTH-1) ||
    (strlen(text2) >= MAX_FILENAME_LENGTH-1) ||
    (strlen(text3) >= MAX_FILENAME_LENGTH-1)) {
    error("string too long"); return;
}
if (outcome == -1) {
    error_1("not a valid blurb command", command);
    return;
}
if (syntaxes[outcome].deprecated) {
    error_1("this Blurb syntax is no longer supported", syntaxes[outcome].explicated);
    return;
}
```

This code is used in §7.

§**9.** The command is now fully parsed, and is one that we support. We can act.

⟨Take action on the command 9⟩ ≡

```
switch (outcome) {
    case author_COMMAND:
        set_placeholder_to("AUTHOR", text1, 0);
        author_chunk(text1);
        break;
    case auxiliary_COMMAND: create_auxiliary_file(text1, text2); break;
    case base64_COMMAND:
        request_2(BASE64_REQ, text1, text2, FALSE); break;
    case copyright_COMMAND: copyright_chunk(text1); break;
    case cover_COMMAND: ⟨Declare which file is the cover art 10⟩; break;
    case css_COMMAND: use_css_code_styles = TRUE; break;
    case ifiction_file_COMMAND: metadata_chunk(text1); break;
    case ifiction_COMMAND: request_1(IFICTION_REQ, "", TRUE); break;
    case ifiction_public_COMMAND: request_1(IFICTION_REQ, "", FALSE); break;
    case interpreter_COMMAND:
        set_placeholder_to("INTERPRETERVMIS", text2, 0);
        request_1(INTERPRETER_REQ, text1, FALSE); break;
    case picture_COMMAND: picture_chunk(num1, text1); break;
    case placeholder_COMMAND: set_placeholder_to(text1, text2, 0); break;
    case project_folder_COMMAND: strcpy(project_folder, text1); break;
    case release_COMMAND:
        set_placeholder_to_number("RELEASE", num1);
        release_chunk(num1);
        break;
    case release_file_COMMAND:
        request_2(COPY_REQ, text1, get_filename_leafname(text1), FALSE); break;
    case release_file_from_COMMAND:
        request_2(RELEASE_FILE_REQ, text1, text2, FALSE); break;
    case release_to_COMMAND:
        strcpy(release_folder, text1);
        ⟨Make pathname placeholders in three different formats 11⟩;
        break;
    case release_source_COMMAND:
        request_3(RELEASE_SOURCE_REQ, text1, text2, text3, FALSE); break;
    case solution_COMMAND: request_1(SOLUTION_REQ, "", TRUE); break;
    case solution_public_COMMAND: request_1(SOLUTION_REQ, "", FALSE); break;
    case sound_COMMAND: sound_chunk(num1, text1); break;
    case source_COMMAND: request_1(SOURCE_REQ, "", TRUE); break;
    case source_public_COMMAND: request_1(SOURCE_REQ, "", FALSE); break;
    case status_COMMAND: strcpy(status_template, text1); strcpy(status_file, text2); break;
    case status_alternative_COMMAND: request_1(ALTERNATIVE_REQ, text1, FALSE); break;
    case status_instruction_COMMAND: request_1(INSTRUCTION_REQ, text1, FALSE); break;
    case storyfile_include_COMMAND: executable_chunk(text1); break;
    case storyfile_leafname_COMMAND: set_placeholder_to("STORYFILE", text1, 0); break;
    case template_path_COMMAND: new_template_path(text1); break;
    case website_COMMAND: request_1(WEBSITE_REQ, text1, FALSE); break;
    default: error_1("***", command); fatal("*** command unimplemented ***\n");
}
```

This code is used in §7.

§**10.** We only ever set the frontispiece as resource number 1, since Inform has the assumption that the cover art is image number 1 built in.

⟨Declare which file is the cover art 10⟩ ≡

```
set_placeholder_to("BIGCOVER", text1, 0);
cover_exists = TRUE;
cover_is_in_JPEG_format = TRUE;
if ((text1[strlen(text1)-3] == 'p') || (text1[strlen(text1)-3] == 'P'))
    cover_is_in_JPEG_format = FALSE;
frontispiece_chunk(1);
char *leaf = get_filename_leafname(text1);
if (strcmp(leaf, "DefaultCover.jpg") == 0) default_cover_used = TRUE;
if (cover_is_in_JPEG_format) strcpy(leaf, "Small Cover.jpg");
else strcpy(leaf, "Small Cover.png");
set_placeholder_to("SMALLCOVER", text1, 0);
```

This code is used in §9.

§**11.** Here, `text1` is the pathname of the Release folder. If we suppose that `cblorb` is being run from Inform, then this folder is a subfolder of the Materials folder for an I7 project. It follows that we can obtain the pathname to the Materials folder by trimming the leaf and the final separator. That makes the `MATERIALSFOLDERPATH` placeholder. We then set `MATERIALSFOLDER` to the name of the Materials folder, e.g., "Spaceman Spiff Materials".

However, we also need two variants on the pathname, one to be supplied to the Javascript function `openUrl` and one to `fileUrl`. For platform dependency reasons these need to be manipulated to deal with awkward characters.

⟨Make pathname placeholders in three different formats 11⟩ ≡

```
set_placeholder_to("MATERIALSFOLDERPATH", text1, 0);
int k = strlen(text1);
while ((k>=0) && (text1[k] != SEP_CHAR)) k--;
if (k>0) { *(read_placeholder("MATERIALSFOLDERPATH")+k)=0; k--; }
while ((k>=0) && (text1[k] != SEP_CHAR)) k--; k++;
set_placeholder_to("MATERIALSFOLDER", text1 + k, 0);
char *L = read_placeholder("MATERIALSFOLDER");
while (*L) { if (*L == SEP_CHAR) *L = 0; L++; }
qualify_placeholder("MATERIALSFOLDERPATHOPEN", "MATERIALSFOLDERPATHFILE",
    "MATERIALSFOLDERPATH");
```

This code is used in §9.

§**12.** And here that very "qualification" routine. The placeholder `original` contains the pathname to a folder, a pathname which might contain spaces or backslashes, and which needs to be quoted as a literal Javascript string supplied to either the function `openUrl` or the function `fileUrl`. Depending on the platform in use, this may entail escaping spaces or reversing slashes in the pathname in order to make versions for these two functions to use.

```
void qualify_placeholder(char *openUrl_path, char *fileUrl_path, char *original) {
    int i;
    char *p = read_placeholder(original);
    for (i=0; p[i]; i++) {
        char oU_glyph[8], fU_glyph[8];
        sprintf(oU_glyph, "%c", p[i]); sprintf(fU_glyph, "%c", p[i]);
        if (p[i] == ' ') {
            if (escape_openUrl) sprintf(oU_glyph, "%%2520");
            if (escape_fileUrl) sprintf(fU_glyph, "%%2520");
        }
        if (p[i] == '\\') {
            if (reverse_slash_openUrl) sprintf(oU_glyph, "/");
            if (reverse_slash_fileUrl) sprintf(fU_glyph, "/");
        }
        append_to_placeholder(openUrl_path, oU_glyph);
        append_to_placeholder(fileUrl_path, fU_glyph);
    }
}
```

The function qualify_placeholder is.