The Implementation of Shsub: A Shell Template Engine Written in C

DONG Yuxuan <https://www.dyx.name>

29 Jul 2023 (+0800)

Shsub is a template engine of the Shell language, implemented in C. This paper explains the implementation of Shsub 2.0.1 to help potential contributors understand the code quickly.

The code of Shsub is hosted at GitHub <https://github.com/dongyx/shsub>.

Introduction

The following file `notes.html.tpl` demonstrates a simple template:

```
<ul>
<%for i in notes/*.md; do-%>
<%      title="$(grep '^# ' "$i" | head -n1)"-%>
        <li><%="$title"%></li>
<%done-%>
</ul>
```

Calling `shsub notes.html.tpl` prints a HTML list reporting titles of your Markdown notes.

The syntax of templates is:

- Shell commands are surrounded by `<%` and `%>` and are compiled to the commands themselves;
- Shell expressions are surrounded by `<%=` and `%>` and each `<%=expr%>` is compiled to `printf %s expr`;
- Ordinary text is compiled to the command printing that text;
- A template can include other templates by `<%+filename%>`;
- If `-%>` is used instead of `%>`, the following newline character will be ignored;
- `<%%` and `%%>` are compiled to literal `<%` and `%>`.

Lexical Analysis

While splitting tokens from the input stream, 3 characters are required to determine if there is a keyword. Those characters may be pushed back to the stream in some cases. However, the standard function `ungetc()` allows only one pushed-back character. Thus a custom stack is defined. It's called the character stack, or, `cstack`.

```
int cstack[3], *csp = cstack, lineno = 1;

int cpush(int c)
{
        if (c == '\n')
```

```
                --lineno;
        return *csp++ = c;
}

int cpop(FILE *fp)
{
        int c;

        if (csp == cstack)
                c = fgetc(fp);
        else
                c = *--csp;
        if (c == '\n') {
                if (lineno == INT_MAX)
                        err("Too many lines");
                ++lineno;
        }
        return c;
}
```

The `csp` variable points to the top of the character stack. The `cpush()` function pushes a character to the character stack. The `cpop()` functions pops a character from the character stack if the character stack is not empty; otherwise, it reads the character from the stream. These functions are also responsible for maintaining the `lineno` variable which is used in syntax error reporting.

There are 8 types of token in Shsub and they are defined by the `token` enumeration. Meanings of these token types are listed in the following table.

Token	**Lexeme**
`INCL`	`<%+`
`CMDOPEN`	`<%`
`EXPOPEN`	`<%=`
`CLOSE`	`%> -%>`
`ESCOPEN`	`<%%`
`ESCCLOSE`	`%%>`
`LITERAL`	non-keyword text
`END`	the sentinel for the end of the input stream

`%>` and `-%>` are combined to a single type `CLOSE`. This makes the grammar simpler that only one token looked ahead is required to determine which grammar rule should be applied. The grammar will be discussed in later text.

Most lexical analyzers can be implemented by a state machine. Shsub doesn’t build the state machine explicitly, but expresses the same procedure in the control flow for brevity and readability. The `gettoken()` function returns the next token type from the stream, and if it’s `LITERAL`, the content is stored to the `literal` buffer.

```
#define MAXLITR 4096

char literal[MAXLITR];

enum token gettoken(FILE *fp)
```

```
        {
                char *p;
                int c, h, r, trim = 0;
                enum token kw = END;

                p = literal;
                while (kw == END) {
                        while (p - literal < MAXLITR - 1) {
                                if ((c = cpop(fp)) == EOF || strchr("<%-", c ))
                                        break;
                                *p++ = c;
                        }
                        *p = '\0';
                        if (c == EOF)
                                return p > literal ? LITERAL : END;
                        if (p == literal + MAXLITR - 1)
                                return LITERAL;
                        h = cpop(fp);
                        r = cpop(fp);
                        if (c == '<' && h == '%') {
                                kw = CMDOPEN;
                                if (r == '=')
                                        kw = EXPOPEN;
                                else if (r == '+')
                                        kw = INCL;
                                else if (r == '%')
                                        kw = ESCOPEN;
                        } else if (c == '%' && h == '>')
                                kw = CLOSE;
                        else if (c == '%' && h == '%' && r == '>')
                                kw = ESCCLOSE;
                        else if (c == '-' && h == '%' && r == '>') {
                                trim = 1;
                                kw = CLOSE;
                        } else {
                                cpush(r);
                                cpush(h);
                                *p++ = c;
                                *p = '\0';
                        }
                }
                if (p > literal) {
                        cpush(r);
                        cpush(h);
                        cpush(c);
                        return LITERAL;
                }
                if (kw == CMDOPEN || kw == CLOSE && !trim)
                        cpush(r);
                if (trim && (c = cpop(fp)) != '\n')
                        cpush(c);
                return kw;
        }
```

The `literal` buffer has a fixed size. If a piece of literal text is too long, it will be split to multiple tokens. This approach avoids dynamic resizing and it won't limit the input length if we design a proper grammar.

The semantic of `-%>` is trimming the following newline character. The trimming is directly executed in `gettoken()` instead of in the subsequent process. Because the semantic is regarding characters instead of tokens, it's more natural to handle it in lexical analysis.

Parsing and Translation

The grammar of Shsub templates is very simple and is described by the following BNF.

```
<tmpl>   :=      <END>
                 <text><tmpl>
                 <CMDOPEN><text><CLOSE><tmpl>
                 <EXPOPEN><text><CLOSE><tmpl>
                 <INCL><LITERAL><CLOSE><tmpl>

<text>   :=      <END>
                 <LITERAL><text>
                 <ESCOPEN><text>
                 <ESCCLOSE><text>
```

There are two non-terminals: *tmpl* and *text*. The *tmpl* non-terminal represents a template. The *text* non-terminal represents a token sequence of literal text or text shall be escaped to literal text.

The filename in an including directive could only be a single `LITERAL`. This limits the length of the included filename, and forbids the filename to contain Shsub keywords. The limit decreases the complexity of the implementation at the acceptable cost: It's rare that a file has a path exceeding 1000 characters or has a name containing '<%', '%>', and etc..

The global variable `lookahead` is used to store the current token. The function

```
void tmpl(FILE *in, FILE *ou)
```

parses *tmpl* from `in` and prints the compiled script to `ou`. The function

```
void text(int esc, FILE *in, FILE *ou)
```

parses *text* from `in`, optionally escaping the text to the shell string representation, and prints the result to `ou`. The function

```
void match(enum token tok, FILE *fp)
```

checks if `lookahead` is matched the desired token and moves `lookahead` to the next token.

```
enum token lookahead;

void tmpl(FILE *in, FILE *ou)
{
        char *p;

        while (lookahead != END)
                switch (lookahead) {
                case INCL:
                        /* Handle template including */
                case CMDOPEN:
                        match(CMDOPEN, in);
                        text(0, in, ou);
                        fputc('\n', ou);
                        match(CLOSE, in);
                        break;
                case EXPOPEN:
                        fputs("printf %s ", ou);
                        match(EXPOPEN, in);
```

```
                                text(0, in, ou);
                                fputc('\n', ou);
                                match(CLOSE, in);
                                break;
                        case LITERAL: case ESCOPEN: case ESCCLOSE:
                                fputs("printf %s '", ou);
                                text(1, in, ou);
                                fputs("'\n", ou);
                                break;
                        default:
                                parserr("Unexpected token");
                        }
        }

        void text(int esc, FILE *in, FILE *ou)
        {
                char *s;

                for (;;)
                        switch(lookahead) {
                        case LITERAL:
                                if (!esc)
                                        fputs(literal, ou);
                                else
                                        for (s = literal; *s; ++s)
                                                if (*s == '\'')
                                                        fputs("'\\''", ou);
                                                else
                                                        fputc(*s, ou);
                                match(LITERAL, in);
                                break;
                        case ESCOPEN:
                                fputs("<%", ou);
                                match(ESCOPEN, in);
                                break;
                        case ESCCLOSE:
                                fputs("%>", ou);
                                match(ESCCLOSE, in);
                                break;
                        default:
                                return;
                        }
        }

        void match(enum token tok, FILE *fp)
        {
                if (lookahead != tok)
                        parserr("Lack of expected token");
                lookahead = gettoken(fp);
        }
```

The `tmpl()` and `text()` functions constitute a recursive-descent parser, but recursive calls are transformed to iterations. The code handling including in `tmpl()` is omitted. Recursive calls are used for that.

Including

When an including directive is scanned, Shsub saves the related global variables to a stack called the including stack, or, `istack`, and recursively parses the included file. Each frame of the including stack is called an including frame, or, `iframe`.

```
	#define MAXINCL 64

	struct iframe {
		FILE *in;
		int lookahead, cstack[3], *csp, lineno;
		char *tmplname;
	} istack[MAXINCL], *isp = istack;

	void ipush(FILE *in)
	{
		if (isp == istack + MAXINCL)
			err("Including too deep");
		isp->lookahead = lookahead;
		memcpy(isp->cstack, cstack, sizeof cstack);
		isp->csp = csp;
		isp->in = in;
		isp->tmplname = tmplname;
		isp->lineno = lineno;
		++isp;
	}

	FILE *ipop(void)
	{
		--isp;
		lookahead = isp->lookahead;
		memcpy(cstack, isp->cstack, sizeof cstack);
		csp = isp->csp;
		tmplname = isp->tmplname;
		lineno = isp->lineno;
		return isp->in;
	}
```

Most elements of `iframe` have been explained in former text, except:

- `in`: The file pointer of the current input file;
- `tmplname`: The filename of the current input file.

The `isp` variable points to the top of the including stack. The `ipush()` function pushes the current context to the including stack. The `ipop()` functions recovers the context from the including stack.

The omitted part of the `tmpl()` function can be completed now.

```
	void tmpl(FILE *in, FILE *ou)
	{
		char *p;

		while (lookahead != END)
			switch (lookahead) {
			case INCL:
				match(INCL, in);
				if (lookahead != LITERAL)
					parserr("Expect included filename");
				if (!(p = strdup(literal)))
					syserr();
				match(LITERAL, in);
				match(CLOSE, in);
				ipush(in);
				tmplname = p;
				if (!(in = fopen(tmplname, "r")))
					err("%s: %s",
						tmplname, strerror(errno));
```

```
                                lineno = 1;
                                csp = cstack;
                                lookahead = gettoken(in);
                                tmpl(in, ou);
                                fclose(in);
                                free(tmplname);
                                in = ipop();
                                break;
                        /* Handling other tokens
                         * ......
                         * ......
                         */
                        }
        }
```

Execution

We can’t pipe the script to the shell, because the script itself may read the standard input. Thus Shsub creates a temporary file to save the script. Should it be a named pipe (FIFO) or a regular file?

Shsub chooses regular file. If we use a named pipe, we must have two parallel processes, and some subtle questions arise. If a signal is delivered to Shsub, should it be forwarded to the shell process? If the shell process terminated, should Shsub aborts parsing and returns the exit status of the shell process? These issues complicate the code.

Shsub attempts to avoid these process and signal issues by calling `execv()` and `execl()` without `fork()`. However, this causes a new issue: How do we clean up the temporary file if we substitute the process image? The solution of Shsub is prepending a command to the script that deletes itself.

```
        char script[] = "/tmp/shsub.XXXXXX";

        int main(int argc, char **argv)
        {
                FILE *in, *out;
                char *sh = "/bin/sh";
                int fd;

                /*
                Misc tasks
                ......
                ......
                */
                if ((fd = mkstemp(script)) == -1 || atexit(rmscr))
                        syserr();
                if (!(out = fdopen(fd, "w")))
                        syserr();
                fprintf(out, "#!%s\nrm %s\n", sh, script);
                tmpl(in, out);
                if (fchmod(fd, S_IRWXU) == -1)
                        syserr();
                if (fclose(out))
                        syserr();
                if (argc > 0)
                        execv(script, argv);
                else
                        execl(script, "-", NULL);
                syserr();
                return 0;
        }
```

The `main()` function also registers a clean-up function `rmscr()` with `atexit()`. If a parsing error happens after the temporary file is created and before the script is executed, `rmscr()` will delete the temporary file.

Building and Installation

One interesting thing about Shsub is that the installation script of Shsub uses Shsub itself.

Shsub 2.0.0 is incompatible to earlier versions. To make parallel installations possible, the makefile of Shsub supports the `name` variable for the `install` pseudo target. The following command installs Shsub with the name `shsub2`.

```
$ sudo make install name=shsub2
```

The `name` variable doesn't only affect the name of the installed files but also the words used in the man page. Thus the man page of Shsub is a Shsub template which is used to generate a real man page with specific `name` during installation.

```
all: shsub

install: all
	name='$(name)' ./shsub shsub.1.tpl >shsub.1
	mkdir -p $(bindir) $(mandir)/man1
	$(INSTALL) shsub $(bindir)/$(name)
	$(INSTALL) -m644 shsub.1 $(mandir)/man1/$(name).1

shsub: shsub.c
	$(CC) $(CFLAGS) -o $@ $<
```

Notice that there isn't a rule of `shsub.1`, because a makefile rule can't depend on a variable. The real man page is always regenerated while installation.