Skel

Shane Lopez

Introduction

Skel is an experimental “pseudo-language” which I have designed for the purpose of program prototype, or “skeleton” generation. It is simple, doesn't require any special diligence beyond proper syntax, and is portable to any machine with a C/C++ compiler. Skel is both a program and a language; one may think of the executable “skel” as a compiler of sorts, and the submitted file the code. By using simple shorthand syntax, Skel can generate a C file with a number of lines far greater than the number of Skel code lines.

Motivation

Actually, Skel began with a tweet from an old room mate of mine who has since gone on to greater things in the computer science world. I had tweeted excitedly about some small triumph of mine in programming, and his tweeted (twittered?) response was:

“*Have you written any functions that can write functions yet?*”

That got me thinking. Why on earth would I want to? Could I? And so I began programming with the inkling of an idea, trying to write a function that could write functions. I succeeded, and had an executable which would write hard-coded functions to a text file. Which was a neat trick, but then I thought, if I can do functions, how about a program?

That took some tinkering, after which I had an executable that could write a hard-coded program to a text file. This was cool the first few times I ran it, but quickly became stale. It was, in a word, useless. If I wanted a program written for me, I essentially had to code it myself, and even then I still had to recompile my code, run it, and compile the generated text file to get it in executable form. And so, wanting more, I picked up a fork().

I did the typing in one forked child, the compiling in another, and finally the execution in the

parent, which worked marvelously – apart from the fact that I was still doing all of the work for my generated program. And so, after all this post-tweet exploration, I decided to make my program a little more useful, which led directly to the first true Skel.

Initial Design

In a previous incarnation, Skel was an executable that required three files to function; one to designate libraries to include in a generated program, another for functions, and finally a file specifying the location of the other two (LIBS, FNCS, and CONF respectively). Using the aforementioned forking procedure, Skel would first scan the CONF file looking for four lines; one that said LIB_F and the line following that one, and a line reading FN_F as well as the line following that one. Then, It would interpret the lines following those two key phrases as file names, scan through the library file (LIB_F) for two other key phrases (STD and CST for standard and custom libraries, respectively), write those library includes to a text file, and follow a similar procedure with the function file, in which went the headers of functions desired by the user (in standard C format).

Following the writing of these functions to the output file, Skel would attempt to compile what it had written, and if that was successful, execute the output file. In a way the additional files made things more adaptable, as there was no need to recompile anything to change the output file- it was as simple as modifying FNCS and LIBS and running Skel again. However, the extra files were clunky and annoying to modify repeatedly, so I came up with a few simple rules for a language of sorts which Skel would be able to interpret, cutting the number of files required down to one; a code file.

Second Design

For round two with Skel, I had some choices to make; mainly, what symbols or key phrases to use, and how to keep things simple. I believe I have succeeded in the latter with my choices for the former, and before going on, I would like to share with you the reader the very small set of rules which

presently dictate Skel's behavior:

1. **Library Includes**

   Libraries can be standard or custom, and are designated by Skel syntax as such:

   #: standard_lib_name !: custom_lib_name

   These may appear anywhere in the Skel code, no need to put them up top. In the first Skel, these were simply written as library names under the flags STD and CST.

2. **Function Declarations**

   While function declarations in the first Skel were straight C, in the new Skel they are a little different. Function declarations can either be extended or abbreviated, though the extended form is intended for use with features not yet implemented in Skel. The two forms follow:

   Extended

   @ function_name, function_type, (function_parameters):

   .

   .

   .

   @!

   Abbreviated

   @ function_name, function_type, (function_parameters):;

3. **Variables and Return Types**

   When declaring the return type of a function, use one of the following values:

   %i for int

   %d for double

   %f for float

   %c for char

   nil for void

   When declaring parameters for a function, use one of the above types (excluding nil) followed by a variable name. Don't forget to separate function parameters with commas!

4. **Forget Main, Man**

   In the interest of time, and because Skel is only for skeleton generation, a standard main function will be written into your generated C file after your declared functions, with a friendly little printf() statement.

And voila, you know the rules for Skel code! As you can see, they are simple and easy to recall. I developed them with quick typing in mind, and tried to save as many characters as I could. Of course, these rules are only for the current version of Skel, and are subject to change in subsequent releases. This is how Skel (currently) works:

1. First, Skel takes argv[1] as a file name and attempts to open it for reading.
2. If the file was found and opened successfully, Skel begins step one of it's parsing procedure, looking for library identifiers. Skel saves information about each library in a special struct I defined for representing libraries, saving all such structs in a vector for

later typing; this is how I was able to specify them anywhere in the Skel code.

3. Next, Skel searches for function identifiers, and like it did with the libraries, saves information about each in a specially defined function struct. Following population, these are all saved in a vector.
4. Now, the aforementioned forking procedure begins, with the typing in one child, the compiling in another, and finally execution in the parent.

In some ways, Skel acts like a compiler, scanning through text looking for specific character sequences and parsing through the information in between to generate something more (which it technically *does* compile). While I was modifying the original Skel code to make it as it is now, there were a few challenges, and even now there are slight issues I would like to address in anticipation of future functionality. I am hesitant to call Skel a language because it is so simple in it's present state, and nowhere near as large or featured as Java or C++; However, I hope to see it evolved in the future, as I really believe it's a neat project.

Pros and Cons

Between the first and second design, I most certainly prefer the second, as the first design was far too high maintenance when it came to small changes in the output. Not that it's a big deal to modify a small text file, but doing two for a common goal is time wasted. Also, function declarations were verbatim C, which might have been nice because there was the possibility of writing standard C operations into them and having Skel type them as such, but was unoriginal and if anything just added time to the typing process.

However, Skel also has a few cons, but these are not insurmountable; for example, it was mentioned to me that using the '%' character to denote a type was less efficient than actually typing out the type since it required two keys followed by a letter, which made '%i' equivalent to 'int' in the

number of keystrokes. To this I can only say that Skel is an ongoing project under development, and perhaps in a later version '%' will have been changed to another symbol requiring only one key, the most appealing to my mind presently being '/'. But, one must always think ahead when developing rules for a language; if I use '/' for variables now, that takes it off the table for comments, structs, operations, and other things down the road.

Overall, I would say that I have had a fun time developing Skel, and intend to work more on it in the future. I have also learned from the experience, and have a greater appreciation for the work of language developers everywhere. Furthermore, I am always open to suggestions, criticisms, witticisms, and general chit-chat regarding Skel, and I hope to get some user involvement in the project. All users are free at any time to modify Skel however they wish, as it is intended to be a tool of use, and different people sometimes need different tools.

The Future of Skel

The future of Skel will include rules for struct definition and simple operations such as looping and basic arithmetic, and may even leverage Perl for some parsing tasks. And, as I mentioned before, Skel is not without it's issues; for one thing, even though structs and operations have not yet been implemented, if and when they are the development will require a bit of modification to the source to deal with things like ordering- as in, function B must come after function A if B calls A, or struct C must be defined before struct D because D has a member of type C inside.

Any number of other issues could crop up while making those changes. However, as a personal project I find Skel stimulating, and look forward to further developing it. Here are a few of my current ideas for future features of Skel:

1. Loops
2. Basic Arithmetic

3. Variable declaration / instantiation
4. Struct/Class definitions
5. Function calls
6. Header file generation
7. An explicitly defined commenting protocol (as it stands, any line without a special sequence looked for by Skel is a comment, which may change or stay as is)
8. Improved parsing (currently Skel relies on each command being on a separate line, I would like to make this a non-issue).
9. Maybe, just maybe, a GUI (I'm not entirely sold on the idea).

For all of these, the main issue will be interleaving each new feature with all of the previous ones in a way that works and does so with minimal effort on the part of the user. And although nothing may come of this project, it will be enough for me if at least one person finds Skel interesting or useful in some small way. If nothing else, it will have been an entertaining personal research project, and fodder for a soon to be sent belated reply tweet.

Skel Tutorial

Given the following C code, can you generate the corresponding Skel?

```
#include <stdio.h>

#include “some_other_header.h”


int returnFour()

{

        int tmp;


        return tmp;

}


double gimmeFive(int a, double b)

 {

        float tmp;


        return tmp;

}


int main(int argc, char** argv)

{

        printf(“\nHello, Master!\n\n”);


        return 0;

}
```

First, let us begin with the libraries up at the top, “stdio.h” and “some_other_header.h”; from the use of the inequality symbols for “stdio.h” and the quotation marks for “some_other_header.h”, we ascertain that the first is a standard library, and the second is a custom one. Recall that standard library includes in Skel take the form:

#: std_lib_name

Custom library includes take the form:

!: cst_lib_name

So, we are left with

#: stdio.h

!: some_other_header.h

Remember, these do not have to be at the top of Skel code.

Next, functions; Recall that the abbreviated (and most convenient) form for a function declaration in Skel is:

@ fn_name, fn_type, (fn_params):;

Further, recall the types:

%i for int

%d for double

%f for float

%c for char

nil for void

Which yields, for function declarations,

@ returnFour, %i, ( ):;

and

@ gimmeFive, %d, (%i a, %d b):;

Being that there is no need to define main in Skel, and that the above C corresponds to the standard main provided, the full Skel specification for the C code above comes out to:

#: stdio.h ←*Can be on any line*

!: some_other_header.h ← *Can be on any line*

@ returnFour, %i, ( ):;

@ gimmeFive, %d, (%i a, %d b):;

or

@ returnFour, %i, ( ):;

#: stdio.h ←*Can be on any line*

@ gimmeFive, %d, (%i a, %d b):;

!: some_other_header.h ← *Can be on any line*

or even

@ returnFour, %i, ( ):;

@ gimmeFive, %d, (%i a, %d b):;

#: stdio.h ←*Can be on any line*

!: some_other_header.h ← *Can be on any line*

As you can see, the C skeleton above, numbering 23 lines (formatted nicely) is defined by Skel in 4; more than 5 times less lines. Huzzah!

As an exercise, generate the C code from the following Skel:

@ practice_fn, nil, ( ):;

#: stdio.h

!: practice.h

!: practice2.h

@ someStuff, %c, (%d a, %c v, %i p):;

#: string.h

Let's attack this line by line. The first line of Skel is a function declaration with return type nil and no parameters. So, we write a standard C function of that description (using the name specified, “paractice_fn”):

```
void practice_fn ( )
{

}
```

Painless! The next line is a library include, and referring to the previous example (and more than likely some experience) we see that it is a standard library. Shove that at the top!

```
#include <stdio.h>

void practice_fn ( )
{

}
```

The next two lines are also library includes, and are custom; again, to the top with these (but below the previous library)!

```
#include <stdio.h>
#include “practice.h”
#include “practice2.h”

void practice_fn( )
{

}
```

Next, another function- return type char and with three parameters (double a, char v, int p):

```
#include <stdio.h>
#include “practice.h”
#include “practice2.h”

void practice_fn( )
{

}

char someStuff(double a, char v, int p)
{
        char tmp;

        return tmp;
}
```

Lastly, a library include, and don't forget the standard main!

```
#include <stdio.h>
#include “practice.h”
#include “practice2.h”
#include <string.h>

void practice_fn( )
{

}

char someStuff(double a, char v, int p)
{
        char tmp;

        return tmp;
}

int main(int argc, char** argv)
{
        printf(“\nHello, Master!\n\n”);

        return 0;
}
```

You are now left with a program skeleton numbering 23 lines produced from Skel code numbering 6 lines- nearly 4 times shorter! As you can see, while the directives are relatively simple, they save the user a bit of typing, and yield a well-formatted (and usually much larger!) skeleton.