Stem

Introduction

Stem is an interpreted concatenative programming language, which is general purpose and features a foreign language interface (FLI), as well as metaprogramming capabilities. Here, I document the syntax a general guide of programming in the language, as well as some of the process of making it. I will also cover adding new functions and objects from the foreign language interface by writing C libraries.

Figure 1: The stem programming language, logo designed by Andrei Sova

If you don't know what any of that means, that is okay. I will go over the programming language as if this is your first programming language, as stem is one of the most simple programming languages that is feasible for practical use. For information on how to install stem on MacOS or Linux, see the github page. Documentation on the C API is available on the Doxygen generated page.

Language Design

In stem, all information is stored on what's called *the stack*, and there are things that you can put on the stack. There is also another type of thing you can do in stem, but that'll have to wait until later. For now, to simplify the explanation, we'll say that *everything that you can do in the programming language stores some information*, and where that information is stored is this thing called *the stack*. With that being said, we will have to define the *what you can do* part and the *stack* part in order for you to be able to program in this language.

Things that can be Stored in the Stack

We call things that can be stored on the stack *literals*. They can be in four different forms, of which two are immediately easy to understand: *strings*, or basically any english phrase or list of characters that you want to store, and *numbers*. Strings look like this:

```
"this is a string!" "1234678876" "this too is a string" "asdfg
```

and numbers look like this:

```
50 3.1415 1000000
```

The third type of literal is called a *quote*. You can imagine a quote as an ordered list of other literals:

```
[ "hello" 50 3.14 [ "inside another quote" ] ]
```

between the '[' and the ']' character, you can see a list of four different literals. Because a quote is also another type of literal, quotes can store other quotes. The *fourth* type of literal we will talk about later, as it is not *just* a literal.

The Stack

Now it is time to talk about the stack. The stack is what stores the literals, of course as we know, but *how* does it store the literals, and for what purpose? It stores the literals like a regular stack of objects, such as a stack of plates, would in real life. When something is put on the stack, it is on the *top* of the stack. When another object is then put on the stack, *that* object becomes the new top of the stack, and the previous object is under that object. This makes a natural ordering of what is considered *above* something else on the stack, just like a stack of plates each with some information on each of them would in real life.

Note that if you had a real life arrangement of these plates, you would be able to read the top piece of information on the stack but no others, until you took that plate off the stack. Then, another plate would be on the top of the stack, and you would be able to read that plate. This is very much like how stem works, but *how* do you read information from the stack, when we've only described how to *put things* on the stack? This is where we introduce the full language: a language of not just literals, but *words* with meaning.

Words

Words are the last type of thing that can be put on the stack, but they are special in that they can also *do things*. Thus far, none of the things we've talked about can actually add numbers, for example, only store them. *Words* add meaning to the language, and make it not just a place to store data, but rather, *do things* with the data. Here are some examples of some words:

```
dsc myword myword123 hello_this_is_word IAMAWORDTOO
```

But most of these words will actually be put on the stack as well rather than do something. In order for them to do something rather than to be interpreted as data, we must *define* them. Stem comes with a set of predefined words that you can combine in order to make new definitions which are defined in terms of a combination of the predefined words, just like in the english language. Next, we'll go over some predefined words.

Predefined Words

To follow along, I suggest after following the instructions on the github page you go into the stem project folder, find the `stemlib` folder, go into it with `cd stemlib`, and then run `stem repl.stem`. Here you will encounter what is known as the *REPL*, or the read, eval, print loop. What it is called doesn't matter. Just know that it runs stem code interactively.

A basic word that prints out the top thing on the stack and removes it is simply a period:

```
# this is a comment, used to explain code but doesn't affect t
# comments start with a '#'.
"hello world\n" .
```

```
hello world
```

where the `\n` just signifies a newline character, basically just telling it to not print the "hello world" on the same line as the next thing printed. You can print the entire stack like so:

```
1 2 3 [ "some quote" ] "string!"

?
```

```
1
2
3
Q: [
some quote]
string!
```

Which prints the entire stack, where the bottom-most thing is the top thing on the stack. There are also some basic math operations you can do:

```
3 4 + .
3 4 - .
3 4 * .
3.0 4 / .
```

```
7
-1
12
0.750000
```

One can independently verify that these results are accurate. These basic math operations take *two* things off of the stack, does the operation on those two numbers, and then puts the new value back on the stack, deleting the old values. Then, the period character prints the value and pops them off the stack.

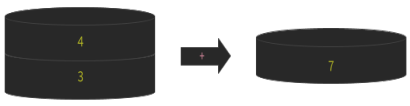

Figure 2: Demonstration of the stack effect of the plus word

There are predefined words for other mathematical operations too, all listed here:

```
0.0 sin .
0.0 cos .
1.0 exp .
2.5 floor .
2.5 ceil .
2.71828 ln .
```

```
0.000000
1.000000
2.718282
2.000000
3.000000
0.999999
```

These operations I will assume you are familiar with, and one can independently verify their (approximate) validity. There are also comparison and logical operations:

```
"hi" "hi" = .
4 3 = .
3 4 < .
3 4 > .
3 4 <= .
3 4 >= .
1 1 and .
1 0 and .
0 1 or .
0 0 or .
```

```
1
0
1
0
1
0
```

```
1
0
1
0
```

Which compare the first number to the second number with a certain operation like “greater than or equals to”. The result is a zero or one, indicating that the statement is either *true* or *false*, with 1 being true. With these statements, you can make decisions:

```
3 4 < [ "3 < 4" . ] [ "3 >= 4" . ] if
```

```
3 < 4
```

where the word `if` just checks if the third thing from the top of the stack (the first thing you write) is a zero or a one, and if it is, then execute whatever’s inside the first quote, otherwise execute the second quote. Note that this wording is a little bit confusing because the *first thing you write* is also the *last thing on the stack* because adding new things to the stack puts the first thing *below* the second.

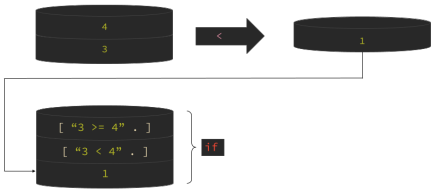

Figure 3: Stack effect of if word

Now, also observe that inside the quotes we are storing valid code. This will become important later on as we introduce the concept of *metaprogramming*. First, though, we have to introduce a couple more important predefined words.

```
[ "hello world!\n" . ] eval
3 quote .
[ 1 2 ] [ 3 4 ] compose .
1 [ 2 3 ] curry .
```

```
hello world!
```

```
Q: [
3
]
Q: [
1
2
3
4
]
Q: [
1
2
3
]
```

`eval` evaluates the top of the stack as if it were a piece of code; `quote` puts the top of the stack in a quote and then pushes it back to the top of the stack; `compose` combines two quotes into one; and `curry` puts a value in the front of the quote. Note that some of these operations work for strings as well:

```
"hello " "world\n" compose .
```

```
hello world
```

And some other words that we use to operate on quotes and strings are here:

```
[ 1 2 3 4 ] 1 cut . .
0 [ 5 6 7 8 ] vat .
"hello\nworld\n" 6 cut . .
1 "asdfghjkl;" vat .
```

```
Q: [
3
4
]
Q: [
1
2
]
5
```

```
world
hello
s
```

`cut` cuts a string or quote into two, where the number in front tells `cut` *where* to cut. Note that normally in programming numbering starts at 0, so 1 is actually the *second* element of the quote. `vat` gets the nth element, where n is the *first* value passed into `vat`. It also returns the quote or string on the stack back after, with the value at that index on top. There are two more words that we have to define:

```
1 2 swap . .
1 2 . .
"hello\n" dup . .
1 2 5 [ + ] dip . .
```

```
1
2
2
1
hello
hello
5
3
```

`swap` just swaps the top two numbers on the stack, `dup` just duplicates the top of the stack, and `dip` is just `eval` except it does the operation one layer below. In this example, it adds 1 and 2 instead of 2 and 5, thus you see a 5 and a 3 printed instead. Note that there are more words, but we won't need them for now. Now, we are ready to investigate how to define words in terms of other words, or so-called *compound words*.

Compound Words

Compound words, or words made up of other words (and literals), are created with yet *another* word, `def`. `def` takes an undefined word (all undefined words are just put on the stack) and a quote, and then from there on the word in question is defined as that quote, where whenever stem sees that word in the future, it immediately `eval`'s that quote. `undef` undefines a word, which is self explanatory.

```
hello [ "hello world\n" . ] def
hello
```

```
\hello undef
hello .
```

```
hello world
W: hello
```

In order to put words on the stack instead of calling them, just escape them:

```
\def .
```

```
W: def
```

Now, so far, we have discussed making decisions with `if`, doing various operations and evaluating quotes in a multitude of ways. What we *haven't* covered is executing the same code some amount of times, or `looping`. In this language, all looping is done by defining words that call themselves, or what's called *recursion*.

Recursion

We can loop in stem by defining a word that calls itself:

```
loop-forever [ "hello world\n" . loop-forever ] def
```

Now, we *don't actually* want to run this because it will just keep on printing hello world forever, without stopping, and we might want to constrain how much it loops. We can do this by only looping under some condition:

```
loop-some [ dup 0 <= [  ] [ dup . 1 - loop-some ] if ] def
4 loop-some
```

```
4
3
2
1
```

and we can see that it actually loops. You can modify the code to do more complex looping, and in the standard library (the `stemlib` folder), there is a `loop` function that loops any code any amount of times,

written by Matthew Hinton.

Metaprogramming

So what is this talk of metaprogramming? To put it simply, metaprogramming is a method by which one can autonomously build code and then evaluate it, thus allowing oneself to talk about code, or make decisions to make different code based on some inputs, before running the code. So how might we use metaprogramming? In the standard library, we define a couple of words `dupd`, `dupt`:

```
dupd [ [ dup ] dip ] def
dupt [ [ [ dup ] dip ] dip ] def
3 2 dupd ?
```

```
3
3
2
```

which duplicates the second and third value on the stack respectively. However, we might want to define `dupn` for any `n`, which takes in an integer and computes `dup n` values down. We can do that with metaprogramming, or less abstractly, we can do it by repeatedly putting quotes inside quotes, and then we can `eval` the resultant quote. Here is the code that programs `dipn` in its entirety, without any sugarcoating:

```
# dsc simply discards the top object on the stack
dsc2 [ dsc dsc ] def
dupd [ [ dup ] dip ] def
over [ dupd swap ] def
dup2 [ over over ] def
dip2 [ swap [ dip ] dip ] def
loop [ dup2 [ swap [ ] if ] dip2 dup [ 1 - loop ] [ dsc2 ] if
dipn [ [ [ dip ] curry ] swap loop eval ] def
dupn [ [ dup ] swap dipn ] def

# this is the code that does stuff
1 2 3 4 5 6 7 8 3 dupn ?
```

```
1
2
3
```

```
4
5
5
6
7
8
```

As you can see, in the early days of programming in this language, you must use quite a lot of words in order to talk about even basic concepts. As the language evolves, however, it becomes ever more easy to "talk" about abstract subjects in it. What this piece of code does is it adds `dip` to the right of the previous quote, nesting quotes like russian dolls over and over again until it becomes suitable to call `eval`. Thus, we have built up a piece of code in the language and then automatically executed it! Note that because `def` is also a word, you can automatically define words as well, which is a powerful concept.

C FLI

The C "FLI" (I don't know what it is really) is a system by which custom objects and functions can be added into the language. A library which implements the bindings for this language have to implement the following functions:

```
void add_funcs();
void add_objs();
```

and has to include the `stem.h` file as a library. The C file has a couple of global variables that it must recognize with extern:

```
extern array_t *STACK;
extern array_t *EVAL_STACK;
extern ht_t *WORD_TABLE;
extern parser_t *PARSER;

extern ht_t *FLIT;
extern ht_t *OBJ_TABLE;
```

not all of these have to be used but it is a complete list of global variables. You may use `add_func` and `add_obj` (defined in stem.h) to add custom functions and custom objects. As long as an object has a specified way to free, deep copy, and create itself, it can be used in a memory safe manner.

The builtin word `clib` loads a library dynamically and calls both `add_func` and `add_obj`, adding all the functions and objects you've defined to the runtime environment. Note that when making custom functions, they must be in the form `void my_func(void *value)`, where `value` contains the word that is used to call the function. When you are implementing functions, make sure to keep only one copy of a pointer on the stack at all times, otherwise it is prone to segfaulting. Also, for examples on how builtins may be implemented, see `builtins.c`.

Again, you can view the API at the webpage generated by doxygen.