home course02 ⇒

Implementing a FORTH virtual machine - 1

course01.c (119 lines) - interpreter

Implements a simple interpreter (no compiler). It provides us with a command prompt, data stack and a dictionary. For now we can't extend the dictionary interactively because we are not able to compile code at this stage. We have some predefined words like + * and hello. We can display which words are there.

Application

ok> is the Forth prompt. One <ENTER> is needed to display the prompt.

```
$ ./course01  # Start application
<ENTER>
ok>
```

Now we have a look at which words are already defined

```
ok> words <ENTER>
. words drop hello * +
ok>
```

We have . (dot) words (already called) drop (which discards top of stack) * (which multiply top of stack with next of stack and leave the result on top of stack. Same with +.

Well, we try to add two numbers

```
ok> 1 <ENTER>
1 ok>
ok> 2 <ENTER>
1 2 ok>
ok> + <ENTER>
3 ok> drop <ENTER>
ok>
```

Or short:

```
ok> 1 2 + <ENTER>
3 ok> drop <ENTER>
ok>
```

As you can see, each item on stack will be displayed.

```
[next-of-stack] [top-of-stack] ok>
```

Source

Now lets talk about the implementation. I only show the important parts, for a more detail view look into the source.

Here we have the main loop. word() discard any leading white spaces and answer with either the next single word to interpret or the null pointer. The nullpointer is the end of file sign.

```
int main() {
        register_primitives();
        while((w=word())) interpret(w);

        return 0;
}
```

Next we have the interpreter which interprets one single word (string of characters without any white space).

```
static void interpret(char *w) {
        if((current_xt=find(w))) { // search the word in dictionary
                current_xt->prim(); // if found, execute it
        } else { // if not found, it may be is a number
                char *end;
                int number=strtol(w, &end, 0); // convert word into number
                if(*end) terminate("word not found"); // not even a number
                else sp_push(number); // push number on data stack
        }
}
```

To find and execute words (which are functions in other languages) each word has an execution token (xt). This token contains the primitive function of the host language (in this case a function pointer to f_dup(), f_words() etc.) In a single list all words are collected.

```
typedef struct xt_t { // Execution Token
        struct xt_t *next;
        char *name;
        void (*prim)(void);
} xt_t;
static xt_t *dictionary;

static xt_t *find(char *w) { // find execution token
        xt_t *xt;
        for(xt=dictionary;xt;xt=xt->next) if(!strcmp(xt->name, w)) return xt;

        return 0; // not found
}
```

To add new words within the host language we maintain only the single dictionary list.

```
static void f_add(void) {
        int v1=sp_pop();
        *sp+=v1;
}
static void add_word(char *name, void (*prim)(void)) {
        xt_t *xt=calloc(1, sizeof(xt_t));
        xt->next=dictionary;
        dictionary=xt;
        xt->name=strdup(name);
        xt->prim=prim;
}
static void register_primitives(void) {
        add_word("+", f_add);
        add_word("*", f_mul);
        add_word("hello", f_hello_world);
        add_word("drop", f_drop);
        add_word("words", f_words);
        add_word(".", f_dot);
}
```

The next step is to add string. Since we have C as the implementation language, we have to do a little bit more to keep strings save in memory.
⇒ course02.html