home ⇐ course02 course04 ⇒

Implementing a FORTH virtual machine - 3

course03.c (269 lines) - compiler

To be able to compile FORTH source we have to introduce some complexity. We need a return stack, a macro dictionary for compiling words, a code segment for the compiled words and an instruction pointer which points to the executing statement.

To execute our compiled code we need a virtual machine as well. The big deal now is that the interpreter and compiler have to run inside the virtual machine. So we have to compile our first word (I name it shell) by hand.

Application

With the word : we define a new word. In this example we define the word *2 and later the word say. All other words until ; are going to be compiled. ; compiles a return statement and switching back from compile mode to interpret mode.
type prints the string (pointer on data stack) on display. cr emits a newline character.

```
ok> : *2 2 * ;
ok> 4 *2
8 ok> drop
ok> : say "so, say what?" type cr ;
ok> say
so, say what?
ok>
```

Implementation

The new interpreter / compiler now works this way:

```
1.1. read next word, finish if imput is empty
1.2. if compile mode => [show at 2.1 compiling]
1.3. if word is a string, push string literal
1.4. else if word is in dictionary, execute word
1.5. else if word is a number, push number on data stack
1.6. throw an error, unknown word

2.1. if word is a string, compile a string literal
2.2. else if word is a macro, execute word
2.3. else if word is in dictionary, compile it
2.4. else if word is a number, compile a number literal
2.5. throw an error, unknwon word

continue at 1.1.
```

lets start with the vm. As you can see, the vm() is an endless loop and does nothing but executing primitives one by one. You might recognize the global variable current_xt which is needed if an execution token hast to reference to itself (see f_docol()). Thus, all the magic relies in the execution tokens.

```
static void vm(void) {
	for(;;) {
		current_xt=*ip++;
		current_xt->prim();
	}
}
```

Now lets see the interpreter / compiler.

code is the current compilation pointer into the code_base segment. Execution tokens (xt_t) and literals get stored there. This is the area for the compiled code.

```c
static void interpreting(char *w) {
        if(is_compile_mode) return compiling(w);

        if(*w=='"') { // +course02: string handling
        // ... code discarded, it is the same as in course02.c
}

static void compiling(char *w) { // course03
        if(*w=='"') { // +course02: string handling
                literal((cell_t)strdup(w+1)); // compile a literal
        } else if((current_xt=find(macros, w))) { // if word is a macro
                current_xt->prim();  // execute it immediatly
        } else if((current_xt=find(dictionary, w))) { // if word is regular
                *code++=current_xt;  // dictionary, compile it
        } else { // not found, may be a number
                char *end;
                int number=strtol(w, &end, 0);
                if(*end) terminate("word not found");
                else literal(number); // compile a number literal
        }
}

static void f_lit(void) {
        sp_push((cell_t)*ip++);
}
static void literal(cell_t value) {
        *code++=xt_lit; // call f_lit when executed
        *code++=(xt_t*)value;
}
```

And now see the functions for : (define a new word, switch to compile mode) and ; (end of definition, switch back to interpret mode)

```
typedef struct xt_t { // Execution Token
        struct xt_t *next;
        char *name;
        void (*prim)(void);
        struct xt_t **data; // address into high level code
} xt_t;

static xt_t *add_word(char *name, void (*prim)(void)) {
        xt_t *xt=calloc(1, sizeof(xt_t));
        xt->next=*definitions;
        *definitions=xt;
        xt->name=strdup(name);
        xt->prim=prim;
        xt->data=code; // current high level code pointer, compilation target
        return xt;
}
static void register_primitives(void) {
   ...
        add_word(":", f_colon); // course03, define new word, enter compile mode

        definitions=&macros; // select macro dictionary
        add_word(";", f_semis); // course03, end of new word, leave compile mode
}

static void f_docol(void) { // course03, VM: enter function (word)new
        rp_push(ip); // at runtime push current ip on return stack
        ip=current_xt->data; // and continue at the high level code
        /* data will be set in add_word() and represent the
        current dictionary pointer */
}
static void f_colon(void) { // course03, define a new word
        char *w=word(); // read next word which becomes the word name
        add_word(strdup(w), f_docol);
        is_compile_mode=1; // switch to compile mode
}
```

```
static void f_semis(void) { // course03, macro, end of definition
        *code++=xt_leave; // compile return from subroutine
        is_compile_mode=0; // switch back to interpret mode
}
```

To get our vm up and running we have to add the shell manually

```
int main() {
        register_primitives();

        /* we compile interpreting by hand */
        add_word("shell",f_docol);// define a new high level word
        xt_t **begin=code;        // save current code pointer for loop back
        *code++=xt_word;          // get the next word on data stack
        *code++=xt_dup;
        *code++=xt_0branch;       // jump to end if top of stack is null
        xt_t **here=code++;       // forward jump reference
        *code++=xt_interpreting;  // interpret/compile word on top of stack
        *code++=xt_branch;        // loop back to begin of this word
        *code++=(void*)begin;     // Loop back address
        *here=(void*)code;        // resolve reference
        *code++=xt_drop;
        *code++=xt_bye;           // leave VM

        ip=begin;                 // set instruction pointer
        vm();                     // and run the vm

        return 0;
}
```

And now lets see how to implement conditionals course04 ⇒