Module 2: Virtual-Machine Loader

Contents

Introduction

This week you'll load virtual object code into your virtual machine, and you'll be able to run it.

- *What am I doing?*
  - Read virtual object code and load it into your VM.
  - Start to learn about *parsing*.
  - Implement some more VM instructions, and continue to apply operational semantics to systems concepts.
- *Why am I doing it?*
  - You're starting to work on two of the main skills you'll get from the course. The first is to implement a programming language in stages, at each stage adding just a little bit to what you already have. Virtual object code adds just a little bit to virtual-machine code.

    The other main skill is parsing, which will be a focus for the next three weeks. You'll be introduced to tokens, and you'll learn about LL(1) parsing techniques, which are both efficient and easy to implement. This is a first step toward designing your own concrete syntax and your own grammars.
  - VM instructions and ability with operational semantics will be needed down the line, when you want to run code with recursive functions.
- *How will I do it?*
  - Before lab you'll study virtual object code and the semantics of loading, and you'll watch an amateur video on LL(1) parsing or learn from other sources. You'll also familiarize yourself with key interfaces and with existing parsing functions.
  - In lab you'll write a parser for the literal values (Booleans, strings, numbers, and so on) that can be mentioned in a machine instruction. And you may start writing a loader.
  - After lab, you'll complete the loader. Then you'll exercise your entire system by implementing some more VM instructions. You should be able to load and run this "hello, world" program:

    ```
    .load module 3
    loadliteral 77 string 14 72 101 108 108 111 44 32 119 111
    print 77
    halt
    ```

    You'll test additional instructions using `check` and `expect`.
  - On Monday night, you'll deliver source code that builds an `svm` binary (with our makefile). And using the `check` and `expect` instructions, you'll deliver two programs, `pass.vo` and `fail.vo` (one is meant to pass a test, and one is meant to fail a test).

The module step by step

The top of the mountain

(1) I have not yet properly introduced vScheme, the language we are ultimately meant to implement. A short description is posted in the "course overview" section of the home page. It can wait a few weeks, so you can skip this step, or you can read it now.

Preparation before lab

(2) Read the handout on virtual object code.

(3) Learn the basics of LL(1), recursive-descent parsing.

- Start with the parsing-background video, which explains you'll learn and why.
- Watch my parsing demo of LL(1) parsing with railroad diagrams. The video discusses most of the grammar for virtual object code.

If you'd rather read than watch, I've curated three different written explanations:

- The **best overall** explanation is section 6.2 of Bob Nystrom's book *Crafting Interpreters*. The opening rant of chapter 6 is pretty good, too. But be aware that *Section 6.1 is the **wrong** way to handle operator precedence and associativity.* By "wrong" I mean that although it works well enough, it gratuitously limits what your language can do with precedence and associativity. If you want a rant on the topic, ask me.

  Nystrom's section 5.1 on context-free grammars is also good.

- If you want something shorter, the **best short** explanation is Ira Baxter's answer to a question asked on Stack Overflow.

- If you want something really short, the **shortest good** explanation is from an old page by Zerksis D. Umrigar at Binghamton University. It's short and sweet, and if you like you can skip the part about "avoiding left recursion," because I've already avoided left recursion on your behalf.

(4) Update your git repository in two steps:

  1. Be sure all your own code is committed. Confirm using `git status`.
  2. Update your working tree by running `git pull` (or you might possibly need `git pull origin main`).

If `git` gives you trouble, please post on the `#git-fu` channel in Slack.

(5) The `git pull` should deliver about 18 new files. You'll need to look at just a few of them. Familiarize yourself with these key interfaces and idioms, which you will need for lab:

- To recognize tokens, you will need the `tokens.h` interface, especially function `first_token_type` and the four `tokens_get` functions.
- To make literal values, you will need the embedding functions in the `value.h` interface, like `mkBooleanValue`. To make strings, you will also need the `VMString_buffer`, `VMString_putc`, and `VMString_of_buffer` functions in the `vmstring.h` interface.
- To encode an instruction that involves literals or global variables, you will need the `eR1U16` function from the `iformats.h` interface. To see an example of how that function is used, look at the `ce0fun` function in file `testfuns.c`.

Lab

(6) Using the `<literal>` nonterminal in the grammar for virtual object code, write a recursive-descent parser for the literal values (Booleans, strings, numbers, and so on) that can be mentioned in a machine instruction. Put your code in file `iparsers.c`. I recommend that you define a function with this prototype:

```
static Value get_literal(Tokens *litp, const char *input),
```

The `litp` argument can be passed to functions like `tokens_get_name`.

If you're not sure how to get started with parsing a token sequence, look at function `loadmodule` in file `loader.c`.

(7) Using the `parseliteral` function you've just written, implement `parseR1LIT`, to parse an instruction that takes one register and a literal value. The new function will closely resemble `parseR1U16`, but instead of reading a 16-bit index, it will read a `.vo` literal. It will then call `literal_slot` to get a 16-bit index.

This parser, which goes in file `iparsers.c`, will be used for instructions like `loadliteral`, `check`, and `expect`.

(8) Now implement `parseR1GLO`, to parse an instruction that takes one register and a global variable. The new function will be almost exactly like `parseR1LIT`, except instead of calling `literal_slot`, it will call `global_slot`.

This parser, which also goes in file `iparsers.c`, will be used for instructions `setglobal` and `getglobal`.

After lab

(9) Continue with your recursive-descent parsing: in file `loader.c`, implement functions `get_instruction` and `loadfun`. Function `get_instruction` reads a single instruction from the `input` file. To read an instruction, you'll read a line, tokenize it, and parse the tokens according to the `<instruction>` nonterminal in the grammar for virtual object code.

(Usually reading an instruction consumes exactly one line, but when `get_instruction` encounters a `.load` directive, it must call `loadfun`, and the call to `loadfun` may consume many lines.)

The `loadfun` function takes a `count` parameter, and it reads instructions from the input by calling `get_instruction` `count` times. The `count` parameter says *exactly* how many instructions are in the loaded function, so `loadfun` can allocate space for them before it starts to read instructions. Protip: *allocate space for one additional instruction* at the end, and fill that space with a `Halt` instruction. This instruction acts as a "sentinel."

At the end of `loadfun`, you'll allocate and return a `VMFunction`. The `arity` is a parameter, the `size` is `count + 1` (the "+1" is for the sentinel `Halt` instructions), and the instructions are what you read. But what about the mysterious `nregs` parameter? That is the number of registers that could be mentioned, which is one more than the largest register mentioned

in the code.[1] That information will be crucial later on, when we implement function calls—we won't want to let a function use more registers than it needs. To compute `nregs`, create a local variable `maxreg`, initialize it, and pass its address to every parsing function. Then `nregs` will be one more than `maxreg`.

(10) If you haven't done so already, implement the `LoadLiteral` opcode in your `vmrun` function.

(11) You can now build your SVM with `make`. Test it using this virtual object code:

```
.load module 3
loadliteral 77 string 14 72 101 108 108 111 44 32 119 111 114 108 100 33 10
print 77
halt
```

You should find that code in file `hello.vo`, and you should be able to run it by running

```
svm hello.vo
```

(12) Your next step is to revisit your `literal_slot` function in file `vmstate.c`. If you took the shortcut of storing every literal in slot 0, you will now have to make it work with more than one literal. There are some space/time tradeoffs here, and I want you to make them thoughtfully. For details, please peek ahead at the learning outcomes for this module.

(13) Now implement function `global_slot` in file `vmstate.c`. The easy path here is to represent the global variables using an association list in the VM state. But the name of a global variable is guaranteed to be a string with no internal zeroes, which gives you more options for your data structures. One interesting option is the `stable.h` interface, which uses an efficient ternary search tree.

(14) While you're editing `vmstate.c`, define functions `literal_value` and `global_name`. Their specifications are in file `vmstate.h`.

These functions are is for use in `disasm.c`; in particular, don't use `literal_value` in `vmrun`.

(15) You'll wrap up the module by building out the set of instructions you can load and run. The loader's parsing is driven by a table in file `instructions.c`, which is indexed by opcode (and opcode name). The same table supports instruction disassembly at run time. Building out the table is your next step; your table needs to support at least 15 VM instructions:

- `halt`, `print`, `loadliteral`, `check`, and `expect`, for which I provide parsers and unparsing templates (5 instructions)
- 3 instructions of your own choice from last week, which you must add to the table
- `getglobal` and `setglobal`, which you must add to the table
- 5 more instructions of your own choice, which you must add to the table

If you want to go all meta, you can design a machine instruction that loads object code from a file, then puts the resulting module value in a register. That's part of a depth goal.

(16) The final coding step is to add the new instructions to your `vmrun`. Make sure all 15 opcodes are recognized by your `vmrun` function, and that they have plausible implementations. (Not all the implementations have to work; as long as you can get the test cases in step (17) to work, broken opcodes won't cost you any points.)

If you have a working add instruction with opcode `+`, you'll be able to run this virtual object code, which I've compiled from the unit test `(check-expect (+ 2 2) 5)`:

```
.load module 6
loadliteral 1 2
loadliteral 2 2
+ 0 1 2
check 0 string 7 40 43 32 50 32 50 41
loadliteral 0 5
expect 0 string 1 53
```

So that you can see if your literal strings are loaded correctly, **I have deliberately set up this test to *fail*.**

Also note: I've assumed the loader will put a halt instruction at the end, so I don't have to write one explicitly.

(17) By hand, write two test cases `pass.vo` and `fail.vo`. Each test case should include a `check` and an `expect`, and each should load successfully. When run, one test should fail and the other should pass.

The tests do not have to be fancy; if you like, you can do everything with just `loadliteral`, `check`, and `expect`. If you want to produce string literals, experiment with the Unix `od` command, as in

```
echo string | od -A n -t d1
```

At this point, it would be reasonable to use `check` and `expect` to test all of the opcodes you've implemented. But coding string literals is tedious, so it would also be reasonable to wait until you have a working assembler that can code the string literals for you. Either way, you're ready to submit.

What and how to submit

(18) On Monday, *submit the homework.* In the `src/svm` directory you'll find a file `SUBMIT`. That file needs to be edited to answer the same sorts of questions you'll answer every week: who did the work, what you're proud of, where code review should focus, and so on. And from this week onward, there are new questions:

- Say if part of your submission includes someone else's code. This includes code you might have gotten from me,

[1] A function that mentions only register 0 mentions 1 register.

unless that code was distributed to everyone as part of the assignment—code that is part of an assignment doesn't have to be acknowledged.

- Say what code, if any, you materially changed in response to code review. Your answer to this question may affect multiple people's participation points, so to get it right, please refer to the discussion in the syllabus.
- Say if you're willing to present at the plenary code review, and if you're not selected to present, what code you would like to see presented. And whether you're willing to serve on the review panel.

Run `make clean`.

To submit, you'll need to copy your working tree to the department servers. We recommend using `rsync`, but `scp` also works.

Now log into a department server, change to the `src` directory of your working tree, and submit your entire `svm` directory:

```
provide cs106 hw02 svm
```

(19) On Tuesday, *submit* your reflection. Create a plain text file `REFLECTION`, which will hold your claims for project points and depth points.

For each project point you claim, write the number of the point, plus whatever is called for in the section "How to claim the project points"—usually a few sentences.

Now copy your `REFLECTION` file to a department server and submit it using `provide`:

```
provide cs106 reflection02 REFLECTION
```

Reading in depth

Occasionally I'll suggest reading that may enrich your view of programming-language implementation.

- *Parsing.* Don Knuth's 1965 paper "On the Translation of Languages from Left to Right"[2] revolutionized parsing. By inventing the class of LR(k) languages, Don brought order and method to what had been a chaotic field. Almost all work since then is based on LR parsing.

  The paper is heavy in parts, but the examples are incisive.

- *Grammars.* Niklaus Wirth's 1977 note on grammar notation observes that "notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation." He proposes we all use EBNF. Alas, Randall Munroe has his number.

- *Linking and loading.* The idea of a textual object code has a long history. Chris Fraser and Dave Hanson describe a machine-independent object code that supports loading *and* linking (that is, resolution of global names before run time), all with multiple instruction formats. A beautiful piece of work that gets at the essence of object code and linking.

Learning outcomes

Outcomes available for points

Learning outcomes for project points:

1. *Instruction table.* You understand the infrastructure well enough to define an instruction table that includes at least 15 different instructions.
2. *Instruction semantics.* You understand the semantics of instructions well enough that your `vmrun` recognizes at least 15 opcodes.
3. *Reuse of language.* Supposing you were asked to implement a language not in the Scheme family, you can say how you would reuse virtual object code.
4. *Reuse of code.* Supposing you were asked to implement a language not in the Scheme family, you can say whether and how you could reuse the loader and the instruction table.
5. *Performance.* You can explain your choices of performance tradeoffs in implementing function `literal_slot`.
6. *LL(1) parsing.* You can identify how alternatives in a grammar correspond to a choice point in a recursive-descent parser.
7. *Invariants.* You can name an invariant that virtual machine code must satisfy but virtual object code need not satisfy.
8. *Embedding.* You understand how the `<literal>` syntax from virtual object code is embedded into the VM `Value` type.
9. *Projection.* You understand which VM instructions can be projected directly into virtual object code.
10. *Formalism.* Your code implements the operational semantics (loader edition).

Just as a reminder, unless they are designated to expire, **depth points can be redeemed at any time during the term.** Some more learning outcomes for depth points are as follows:

11. *Operational semantics [0.25 points].* You can write down a rule of operational semantics to specify what the loader does with the `<module>` form.
12. *Operational semantics [1 point].* The vScheme global environment ρ is meant to be represented by a compositional mapping from names to locations: $\rho = G \circ N$. And in vScheme, the initial basis ρ_0 is a *total* function: every named variable has a location. But in the virtual machine, G is a *partial* function: only finitely many global variables have designated locations

[2] If you can't get through the paywall, let me know and I'll help you out.

in the VM. This discrepancy can be resolved by writing suitable operational semantics, and you can show two different ways to do it (half a point each, partial credit OK).

13. *Tokenization and parsing [0.5 points].* Suppose that we want to change the on-disk representation of object code so that instead of a sequence of integer codes, a literal string is represented as a C-style string literal, with double quotes. To handle the new representation, what has to change in the *interface* described in file `<tokens.h>`?

14. *Dynamic compilation and loading [2 points].* μScheme and vScheme both have a `use` syntactic form, which tells the interpreter to load and run code. This goal is to implement two machine instructions, one to call `popen` and read from a pipe, and one to load a list of modules from an open file descriptor. These two instructions can then eventually be used to call the compiler and load the results. These instructions can be tested now, and then by the time of module 4, they can be used to implement a `use` function.

How to claim the project points

Each of the numbered learning outcomes 1 to 10 is worth one point, and the points can be claimed as follows:

1. Have submitted, by Monday night, an `instructions.c` whose table includes at least 15 different instructions. Each entry must include a parsing function and an unparsing template.

2. Have submitted, by Monday night, a `vmrun.c` whose `vmrun` function recognizes at least 15 different opcodes. Each opcode must have a plausible implementation, but the implementations do not need to have been tested, and don't have to be correct—each implementation just has to look like a good-faith effort to implement an instruction.

3. To claim the project point for this outcome, choose a language not in the Scheme family. Enumerate the forms of literal values that can be written in the language, and for each form, say whether you could express a literal of that form as a `<literal>` in virtual object code, or whether a compiler for your chosen language would have to generate a sequence of VM instructions to materialize the literal. If there are any forms of literal that would have to be materialized using a sequence of machine instructions, pick one and explain how the form would be materialized. Otherwise, pick any literal form and explain how it would be expressed using an object-code `<literal>`.

4. To claim the project point for this outcome, choose the same language you chose for the previous outcome, and say whether you expect to be able to reuse the loader and the instruction table. (I am asking not about the instructions themselves, but about the mechanisms used to list, parse, and encode instructions.) If you think you could reuse the code, say what would have to change, if anything. If you think you could not reuse the code, explain why not.

5. We'd like `literal_slot` to have all of these properties:

   - There is no wasted space in the literal pool; that is, except for `nil`, different slots never contain equal literals.
   - Adding a literal to the pool takes constant time.
   - Implementing `literal_slot` is quick and easy.

   But it might be necessary to compromise. To claim the project point for this outcome, say which of these properties *your* implementation has, and explain why you chose the properties you did.

6. The grammar for virtual object code shows six different forms of `<literal>`. To claim the project point for this outcome, identify the exact lines of code where your parser chooses *which* form of `<literal>` it thinks it sees. Since there are multiple forms of `<literal>`, there might be multiple lines in the code.

7. The VM machine language and the language of virtual object code have very similar forms of instructions. As an invariant, for example, both forms can express an instruction that operates on two registers. But the language of virtual-machine code is more limited: a virtual-machine instruction cannot mention a literal; it can only mention 8-bit, 16-bit, or 24-bit fields. A virtual object code instruction can mention a literal. To claim the project point for this outcome, name an invariant that virtual machine code must satisfy but virtual object code need not.

8. The grammar for virtual object code shows 6 forms of `<literal>`. To claim the project point for this outcome, say which of these 6 forms, if any, can be embedded into a VM `Value`, *assuming that no VM state is available.* If there are any such forms, point to one location in your code that embeds one such form.

9. There are 2^{32} values of type `Instruction`, but for this question we consider only the ones produced using the instruction table in file `instructions.c`. (They are produced using a form of embedding.) Of the instructions listed in your instruction table, which ones can be *projected* into virtual object code matching one of the three `<instruction>` forms, even without access to a VM state? What do these instructions have in common?

10. Two rules of the operational semantics share similar premises: in one case, $L \subseteq L'$ and $L'(k) = V$, and in the other case, $L' \subseteq L''$ and $L''(k) = V$. You can say where in your code those premises are implemented.

VM Instructions: Suggestion of the week

The Standard ML string library contains two nice primitive functions: `explode` and `implode`. Function `explode` takes a string as parameter and returns a list of the characters found in the strong. And `implode` is its inverse. They would make a nice pair of VM instructions for operating on strings—and you could then implement string processing using all the usual list-processing functions,

including the parsing combinators you will develop in modules 3 and 4.

If you want to make really nice versions of these instructions, make them work with UTF-8.