Module 4: Assembly

Introduction

This week you'll finish the parser for your assembly language, and you'll implement the first forward pass in your translator: from assembly language to virtual object code. You'll be able to run hand-written assembly code by translating it to object code and handing the result to your VM. The translation pass is the intellectual meat of the module, but the theme of the week is "let's build infrastructure for debugging."

- *What am I doing?*
  - Complete and debug a parser for your assembly language.
  - Implement the *label-elimination* pass that translates assembly language to virtual object code.
  - Build out your assembly language to support every vScheme primitive.
- *Why am I doing it?*
  - A readable assembly language will help you debug the rest of your Universal Forward Translator. You'll be able to read assembly-language output, and when necessary, edit it by hand and send the results downstream.
  - Parsing and unparsing skills will serve you well even after class is over: They are useful in any situation where data is stored archivally on disk but is operated on in memory.
  - Label elimination introduces you to the technique of structuring a translator as a sequence of simple passes. The pass itself, which captures the essence of assembly code as "machine code plus labels," is commendably simple.
- *How?*
  - In lab you'll develop a fold operation over sequences of assembly instructions, and you'll use it to implement a two-step translation

pass (label elimination) from assembly code to object code. Or if you prefer, you can get help debugging your parser.

– At the end of the module, you'll deliver a working Universal Forward Translator that implements two passes: The `vs-vs` pass parses and unparses assembly language, validating both the parser and the unparser. The `vs-vo` pass turns assembly language into virtual object code, which you can run. You'll also deliver your grammar, some test files written in assembly language, and any updates that are needed to make sure the resulting object code can be loaded into your VM (but not necessarily run).

The module step by step

There are a lot of steps, but many of them are very small.

Before lab

(1) *Download updates.* Update your git repository in two steps:

   a. Be sure all your own code is committed. Confirm using `git status`.

   b. Update your working tree by running `git pull` (or you might possibly need `git pull origin main`). You should not encounter any merge conflicts in `asmparse.sml`. If you do, ask course staff for help.

(2) *Build.* Confirm that you can build a UFT with `make`.

(3) *Learn about languages in the UFT and the SVM.* The course overview includes a video that reviews our low-level languages (assembly code, virtual object code, and virtual machine code). If you haven't watched the video yet, watch it before you decide on a lab.

(4) *Decide on a lab.* Look over the two lab options below, and by 11:00am Thursday, let me know if you prefer option 1 (label elimination) or option 2 (parsing help). If you're not sure, pick option 1 (label elimination)—there will be plenty of parsing help available later in the week.

(5) *Review for your chosen lab.*

   - If you've chosen option 1, review the differences between `AssemblyCode.instr` and `ObjectCode.instr`. Particularly the two forms of `GOTO` and the fact that the `ObjectCode` form does not have any analog to `AssemblyCode.DEFLABEL`.

   - If you've chosen option 2, write down in a grammar the part of the concrete syntax you want help parsing, and review your parsing code.

Lab option 1: Label elimination

Today's lab can get you most of the way to your translation pass. If that seems like a good use of today's time, choose option 1 for lab, with steps (6) to (9). If getting help with your parser seems like a better plan, choose option 2 for lab, with step (10).

Label elimination takes two passes:

- The first pass computes and records the position of every label.
- The second pass replaces assembly-language `GOTO_LABEL`, which branches to a label, with an object-code `GOTO`, which branches relative to the position of the `GOTO`. And it discards the labels.

Everything else just mapping each assembly-language instruction and function into the corresponding object-code instruction and function.

Because each pass has the same overall structure, in step (7) I recommend that you define a `fold` function that can be used for both passes. You can then implement the passes in steps (8) and (9).

(6) *Get ready to code.* In this lab you'll edit file `assembler.sml` in directory `src/uft`. Start by renaming function `translate` to `old_translate`.

(7) *Fold over instruction streams.* We'd like to implement both passes by folding from left to right. But an ordinary fold over a list of instructions gives its argument function just two values: an instruction and an accumulator. To implement label elimination, we also need to know the *position* of every instruction in the instruction stream—that is, the number of instructions that will precede it in the translated object code. To compute this position, define a new, right-to-left[1] `fold` function that operates on lists of type `AssemblyCode.instr list`. The `fold` function applies a function f of *three* arguments:

- The next thing in the list of type `AssemblyCode.instr`
- The *number of machine instructions*, not counting label definitions, that precede the next thing in the list
- An accumulator of unspecified type `'a`

Function f returns a (possibly new) accumulator of type `'a`, so its type is

f `: int * AssemblyCode.instr * 'a -> 'a`

The complete type of `fold` is

```
val fold : (int * AssemblyCode.instr * 'a -> 'a) -> 'a -
> AssemblyCode.instr list -> 'a
```

[1] In a right-to-left fold, the result of a recursive fold is passed to f. In a left-to-right fold, the result of calling f is passed to a recursive fold. Both directions work equally well for `labelEnv`, but the right-to-left is what you want for `labelElim`. (Using a left-to-right for `labelElim` will reverse the list of instructions.)

You'll use your `fold` function in each pass.

To define `fold`, you need to know exactly how much space each *assembly* instruction takes up in the final *object* code:

- A label definition takes no space.
- An `IF_GOTO_LABEL` takes two slots (one for `if` and one for `goto`).
- Every other instruction, including `LOADFUNC`, takes exactly one slot. (The `LOADFUNC` carries many instructions, but they go into a `VMFunction` that is added to the literal pool. As you know from your SVM, only a single load-literal instruction is emitted into the current instruction stream.)

Function `fold` should *not* do any error handling. Leave the error handling to other functions.

Function `fold` can't be recursive! To solve the problem recursively, you have to know the position of each instruction, which changes. But `fold` always starts at position 0. So define a recursive helper function, then define fold by calling the helper with initial position 0.

(8) *Find label definitions.* Use your `fold` function to define this function:

```
val labelEnv : AssemblyCode.instr list -> int Env.env error
```

This function returns an environment that says, for each label, what position in the instruction stream it stands for. The function fails (that is, returns `ERROR` with a message) if any label is defined in more than one place.

To help with `labelEnv`, I found it useful to define a `lift` function that takes care of error handling. It has this type:[2]

```
val lift : ('a * 'b * 'c -> 'c error) -> ('a * 'b * 'c error -> 'c error)
```

I defined `lift` because error handling is hell. I want to write a function `g` that just takes in an `'a`, a `'b`, and a `'c`, and `g` never has to worry about bad input. That makes `g` easy to write. But what if the third input could be bad? Then I can use `lift g`, which is still guaranteed to get a good `'a` and `'b`, but might get a bad `'c`. How does `lift g` work? If it gets a good `'c`, wrapped in `Error.OK`, then it passes three good inputs to `g`. Otherwise it can't call `g`, and it already has a `'c error`, so it just returns it.

I then applied `fold` to `lift` f. And within my f, I used `fail` as a synonym for `Error.ERROR`:

```
val fail = Error.ERROR
```

(9) *Eliminate labels.* Use your `fold` function again to define two mutually recursive functions:

[2] If the type of `lift` baffles you, think about the type of `fold`.

```
val labelElim :
  AssemblyCode.instr list -> int Env.env ->
  ObjectCode.instr list Error.error

val translate :
  AssemblyCode.instr list -> ObjectCode.instr list Error.error
```

Both of these functions return the same value: either a successfully translated list of object-code instructions, or an error message. The only possible error is an attempt to go to a label that is not defined.

The mutual recursion works like this:

- Function `labelElim` calls `translate` when it needs to eliminate labels from the body of a `LOADFUNC` form.
- Function `translate` combines the two passes `labelEnv` and `labelElim`.

This structure ensures that labels within a function's body are private to that function.

Notes:

- To define `labelElim`, I didn't use `lift`; instead, I used `<$>` and `<*>` from the error monad. I used this boilerplate code, which I left in `assembler.sml` for you to use also:

  ```
  type 'a error = 'a Error.error
  val (succeed, <*>, <$>, >=>) = (Error.succeed, Error.<*>, Error.<$>, Error.>=>)
  infixr 4 <$>
  infix 3  <*>
  infix 2  >=>
  ```

- Function `translate` can be defined in one line, but the multiple `error` types make its definition a bit tricky. The argument list is used *twice*, so a simple function composition is not going to work. I wound up using the Kleisli composition arrow `>=>`, then applying the resulting composition to the input list of instructions. There other good ways to do it; work through the types.

Lab option 2: Parser design and construction

(10) *Get help with parsing.* This option allows you to use lab time essentially as extra office hours. You have these options:

- Help with the lab exercises from last week
- Help getting your own parsers to typecheck
- Help getting your own parsers to work
- Help changing or extending the assembly-language lexer.

After lab

Parsing and unparsing

(11) *Create a test file for your parser.* Create file `allsyntax.vs`. Using the concrete syntax that you have designed for your assembly language, this file should include

- At least one example of every instruction that your SVM recognizes
- At least one example of the "goto label" and "if register then goto label" instructions, even if your SVM doesn't recognize them yet
- An example label definition

The code in file `allsyntax.vs` needn't *do* anything; it is meant to test parsing and unparsing.

(12) *Parse what you have so far.* In file `asmparse.sml`, build out your `one_line_instr` parser to the point where everything in `allsyntax.vs` can be parsed without error. You may want to consult my hints for writing and debugging combinator parsers.

A workable approach here is to run

```
uft vs-vs allsyntax.vs > /dev/null
```

and correct whatever issue is reported in the *last* error message. Keep going until there are no more error messages.

As you work, **record your first diagnosed bug and your most challenging bug** for use in the learning outcome on debugging parsers.

(13) *Confirm parsing and unparsing.* Using your UFT, confirm that every instruction in `allsyntax.vs` can be parsed and unparsed successfully. The output should be identical to the input, or possibly have differences in white space. Here is a way to test it at the command line:

```
uft vs-vs allsyntax.vs | diff -b allsyntax.vs -
```

(14) *Implement function loading.* In file `asmparse.sml`, search for "grammar", which has this grammar for instructions:

```
<instruction> ::= <one_line_instruction> EOL
                |  <loadfunStart> {<instruction>} <loadfunEnd>
```

(The `EOL` stands for "end of line" and is how the `\n` character is represented inside the parser.) Observe the `loadfunStart` and `loadfunEnd` parsers: they are *placeholders* that you are meant to replace. Complete this step in three stages:

- Design concrete syntax to mark the start of a "load function" instruction. This syntax must include the destination register into which the function will be loaded, plus the number of arguments that the

function is expecting. If you have already implemented an *unparser* for the `LOADFUNC` form, start with that syntax.

- Implement the `loadfunStart` parser. The marker for a function start can be as simple as a single token or as fancy as you like.
- A "load function" instruction is followed by a sequence of assembly-language instructions, one per line. This sequence should be followed by some sort of closing delimiter, perhaps on a line by itself. Design concrete syntax for that delimiter, and implement it in the `loadfunEnd` parser.

  The delimiter can be as simple as a single token or as fancy as you like. But **it must not look like an instruction.**
- Create a test file `loadfun.vs` that exercises the new syntax.
- Confirm that assembling the file generates a "load function" for the SVM:

  ```
  uft vs-vo loadfun.vs | fgrep .load
  ```
- Confirm that the function loads without error:[3]

  ```
  uft vs-vo loadfun.vs | svm
  ```
- You'll confirm that it loads correctly later, after you've made a small extension to the SVM.

Label elimination

(15) *Deploy the translator from lab.* If you haven't already, in file `assembler.sml`, replace my toy version of function `translate` with a complete one. Follow steps (6) to (9) from the lab, option 1.

A complete instruction set

The final part of the homework is to enrich your *parser*, *unparser*, and *SVM loader* to handle all the primitive functions of vScheme. These functions come in two species: those executed for *value* and those executed for *side effect*. The primitives that are executed for value are as follows:

```
function?   cdr   =   idiv   hash
pair?       car   >   /
symbol?     cons  <   *
number?               -
boolean?              +
null?
nil?
```

[3] We cannot call functions yet, so you cannot test if it runs.

Each corresponding VM instruction needs to put a result value in a register. This need should be reflected in the concrete syntax, as it is in these examples:

```
r1 := r2 + r3
r7 := symbol? r9
```

The primitives that are executed only for side effect do *not* produce a result value in a register:

```
error
printu
print
println
```

For these primitives, I recommend a concrete syntax that simply names the argument, as in

```
print r7
```

But you may do as you wish.

If you are unsure what the primitives do, you can experiment with `vscheme -vv`. (To build the `vscheme` binary, run `make` in directory `src/vscheme`.) Most of these primitives are also primitives of μScheme, so if you have taken or are taking CS 105, you know them already.

This week, *you have to parse and load all these instructions*, but **you don't have to implement them all in `vmrun`.**

Complete these steps:

(16) *Design assembly-language syntax for primitive vScheme functions.* Extend your assembly-language grammar to support vScheme's primitive functions. Put your (extended) grammar, with *all* of your instructions, in file `GRAMMAR`.

For each primitive, add a case to your `allsyntax.vs` file (unless the file has one already).

(17) *Extend your parser.* In file `asmparse.sml`, extend function `one_line_instr`, which you began last week (when it was called `instr`). Handle every instruction in `allsyntax.vs`.

(18) *Extend your unparser.* Any new instructions you added in step (16) needs to be unparsed. Add them to function `unparse` in file `asmparse.sml`.

(19) *Confirm your extended parser and unparser.* The following test should not produce any output:

```
uft vs-vs allsyntax.vs | diff -b allsyntax.vs -
```

Recognizing the instructions in the SVM

(20) *Extend your instruction table.* In your SVM, add entries to `instructions.c` and `opcode.h` so that every new opcode is recognized and has an internal form. It is OK for a single internal form like `Add` to have multiple opcodes in a `.vo` file, like both `add` and `+`. Just create multiple entries in the instruction table.

Confirm that your SVM builds with `make`.

(21) *Confirm instruction loading and SVM disassembly.* In this module, the SVM ships with a new executable binary program `svm-dis`. Use this binary to confirm that the unparsing templates in the instruction table produce good output. You won't match the `if` or `goto` instructions, but the others should be close.

```
uft vs-vo allsyntax.vs | svm-dis | diff -y -W 80 allsyntax.vs -
```

Put at least *one* working instruction in file `round-trip.vs`, and confirm that the input is reproduced exactly. This command should show only the addition of a `halt` instruction (by the loader).

```
uft vs-vo round-trip.vs | svm-dis | diff round-trip.vs -
```

If `round-trip.vs` contains just one instruction, the output should look like this:

```
1a2
> halt
```

(22) *Confirm function loading and SVM disassembly.* You won't be able to compare output line by line: In assembly code and virtual object code, one function's instructions may be interrupted to load another function, but after loading, each function's instructions are contiguous. But you can still eyeball the result and see if the VM loaded a function with the right instructions:

```
uft vs-vo loadfun.vs | svm-dis
```

What and how to submit

(23) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.04`. That file needs to be edited to answer the same questions you answer every week.

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 your working tree, and submit your entire `src` directory:

```
provide cs106 hw04 src
```

(24) 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 reflection04 REFLECTION
```

Reading in depth

Occasionally I'll suggest reading that may enrich your view of programming-language implementation. Here is the original article on *applicative functors*, which defines `succeed` (there called `pure`), `<*>`, and `<$>`:

- Conor Mcbride and Ross Paterson. 2008 (January). Applicative programming with effects Journal of Functional Programming, 18(1):1–13

I regret to say that the examples in the paper are mostly unfamiliar, and that they don't delight me.

Learning outcomes

Outcomes available for points

You can claim a project point for each of the learning outcomes listed here. Instructions about how to claim each point are found below.

1. *Understanding label elimination.* You understand error detection in the label-elimination pass.
2. *Writing test codes.* Your `allsyntax.vs` includes at least one example of each of these instructions from previous modules: `check`, `expect`, conditional, `goto`, get and set global variable, register move (copy), error, load-literal, and halt instructions.
3. *Parsing single-line instructions.* You can explain your `one_line_instr` parser.
4. *Parsing function loads.* You can explain how your parser is intended to handle function loading.
5. *Debugging combinator parsers.* You can explain how you debugged your parser.
6. *Unparsing in the instruction table.* You understand how to use the unparsing template in the SVM's instruction table.

7. *Extending the instruction set.* Your `allsyntax.vs` includes an example of each vScheme primitive listed in step (16).

8. *Embedding and projection.* You understand embedding and projection as it relates to `asmparse.sml`.

9. *Monads.* You understand how to transfer operations between monads.

10. *Combinator parsing.* You understand the producer interface well enough to define new parsing combinators.

You can claim most of a depth point for improving `the`:

11. *Parsing quoted text [0.8 points].* My function `the` works only when the argument given is a single token. Update the function so it works even when the argument has to be represented by a sequence of tokens, like `"!="`.

    (The ability to do stuff like this is a big chunk of why people like parsing combinators.)

    The fractional point expires with this module, i.e., it must be claimed by submitting an updated `the` Monday night.

How to claim the project points

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

1. To claim this point, do two things:

   - In `assembly.sml`, point to a line of code you have written that passes a message to `Error.ERROR`. Explain in source-language terms what error the code reports.
   - Give an example of an analogous error that could occur in a C program or an ML program, that a compiler would be able to detect. A description in informal English is enough; the example need not show source code. And the analogy need not involve labels *per se.*

2. To claim this point, tell us where in your `allsyntax.vs` the examples can be found. "Right at the beginning" would be a good and sufficient answer (if accurate).

3. A claim for this point will have to be supported by an instruction from your `allsyntax.vs`. The computer can find the middle line for you; if you run `sh` or `bash`, you can run this command:

   ```
   sed -n "$(expr $(cat allsyntax.vs | wc -l) / 2)p" allsyntax.vs
   ```

   If that line contains an instruction, use it; if it contains function-loading syntax, use the nearest preceding instruction.

To claim this point, show us the instruction, and identify the lines of code in your `one_line_instr` function in your `asmparse.sml` that are intended to parse the instruction.

4. To claim this point, give us the line numbers of the start and end of the first (and possibly only) function loaded in your `loadfun.vs` file. And explain how your parser in `asmparse.sml` is intended to handle those lines.

5. To claim this point,

   - Identify a part of your assembly-language parser that did not work initially.
   - Identify the line in your test file (`allsyntax.vs`, `loadfun.vs`, or other) that contains an input that exposed the fault.
   - Say what you tried to diagnose or correct the fault and whether your efforts were successful.

   If everything you wrote worked on the first try, you cannot claim this point.

6. This point can be claimed only if you submitted the working `round-trip.vs` that you wrote in step (21) above. To claim the point, identify the entry in the instruction table that contains the relevant unparsing template.

7. To claim this point, tell us where in your `allsyntax.vs` the examples can be found. "At the end" would be a good and sufficient answer (if accurate).

8. The candidates for embedding and projection in `asmparse.sml` are the "unparser," which is understood to mean function `AsmParse.unparse`, and the "parser," which is understood to mean a suitable Kleisli composition of `AsmParse.parse` with `map AsmLex.tokenize` and `Error.list` (as in `uft.sml`). To claim this point,

   - Say whether the parser and unparser constitute an embedding/projection pair.
   - If they are such a pair, which is the embedding and which is the projection?
   - Say how you know the answers to the previous two questions.

9. The parsing monad and error monad actually share more operations than are shown in their interfaces as I provided them. To claim this point, transfer an operation from one monad to the other in one of the following three ways:[4]

   - To claim the point on easy mode, give the type of the choice combinator `<|>` in the error monad, and give algebraic laws that define its behavior.

[4] I claim that each of these transferred operations is useful.

- To claim the point on normal mode, give the type of the `list` function in the parsing monad, and give algebraic laws that define its behavior.
- To claim the point on hard mode, give the type of the monadic bind operation `>>=` in the parsing monad, and define its behavior by giving *either* algebraic laws or ML code.

All three modes are worth the same: one point. Claiming multiple modes does not earn multiple points. Providing three modes give you a chance to discover where you are.

10. To claim this point, define a parsing combinator `commaSeparated` that captures the common syntactic abstraction of "zero or more things separated by commas." Give the type of `commaSeparated`, and define its behavior by giving *either* algebraic laws or ML code.

    If you successfully claimed the point in the previous module, you can't claim it again.