Module 3: Parsing and Unparsing: Assembly Language and Virtual Object Code

Introduction

This week you'll design your assembly language and implement an unparser for it. And you'll start learning about parsing combinators and about the Universal Forward Translator. The parsing stuff is a lot to swallow, but you'll have two weeks to assimilate it; your full parser isn't due until module 4.

The theme for the week is designing concrete syntax that promotes readability.

- *What am I doing?*
  - Learn the representation of assembly code that is used inside the Universal Forward Translator (it's a superset of virtual object code).
  - Design your own concrete syntax for the assembly language.
  - Unparse internal representations into your concrete syntax.
  - Start learning to program with monads: an error monad and a parsing monad.
- *Why am I doing it?*
  - You'll tighten your grip on our layered design strategy for language implementation: assembly language is one more language layer. It adds just two new features: readable concrete syntax plus the ability to use labels.
  - You'll prepare your system to be easy to debug: the concrete syntax you choose will be what you read as you enhance and debug your translator in modules 5 through 10. By designing a syntax that *you* find easy to read, you will help your future self.
  - You'll deepen your parsing skills by generalizing them to the higher-order, monadic setting.
    - Unless parsing performance is critical, monadic functions for parsing will be all you will ever need.

    - Monadic parsers are powerful and fun to use.
    - After the course, you will be able to use monadic parsers for *any* task that involves textual data, not just programming languages. And you won't be dependent on somebody else's parsing library—at need, you'll be able to create your own library of parsing combinators. All you need is !

  - Throughout your translator, a transformation that might fail will return a result in the error monad. That means that unlike with exceptions or assertions, the possibility of failure is explicit in the type. Programming with the error monad will raise your functional-programming game.

- *How will I do it?*
  - You'll spend substantial time studying concepts of error management and combinator parsing. You'll also learn a number of representations and interfaces inside the UFT. You'll write a little bit of code, not a lot.
  - Before lab, you'll be introduced to interfaces for two monads: an error monad and a parsing monad. Because they are both monads, these interfaces have important functions in common. For the error monad, you'll have a short handout; for the parsing monad, you'll have a short video and a longer handout.
  - To prepare for lab, you'll come up with algebraic laws for the "baby error monad." You'll also develop a grammar for a fragment of your assembly language. At this stage your grammar needn't be complete; if it can express a few instructions during lab, that will be enough.
  - In lab, you'll use either code or algebraic laws to implement a couple of functions in the parsing monad. Then you'll use the parsing monad to write a parser for one form of the concrete syntax of your assembly language. That will be all the parsing you need to do this week.
  - After lab, you'll write an *unparsing* function that renders your complete SVM instruction set exactly the way you wish it to appear in your concrete syntax. You'll extend my "escape-hatch" parser for assembly code so that it recognizes the one form you did in lab. And you'll learn how to command the Universal Forward Translator to parse assembly code and to translate it to virtual object code.

The module step by step

Before lab

(1) *Review ML.* Review your knowledge of Standard ML. Or if you are preparing to write the UFT in Haskell or in some other functional language, review that. Especially review pattern matching, Curried function application, lists, and the `option` type with its `SOME` and `NONE` constructors.

(2) *Learn about error monads.* Read about the baby error monad and **complete the algebraic laws.**

We'll look at your laws before other lab work. If you want to maximize your lab time, *DM your laws to me* before lab starts (by 1:00, please).

(3) *Start your syntax design.* Design concrete syntax for a fragment of your assembly language. Following the design guidelines, write a grammar that can express at least these three instructions:

- Add two registers and put the sum in a third
- Load a literal value in a register
- Print a register

(4) *Learn what parsing combinators can do for us.* To learn what problems parsing combinators solve for us, you have your choice of two videos. Each will give you a different perspective. Watching both would be reasonable, but a single one suffices.

- Watch a my short demonstration video made for you.
- Watch a lightning introduction made for a professional audience.

The professional video, which starts at 28:14, needs a little explanation:

- Although higher-order functions have been used for parsing for almost thirty years, new techniques are still being developed. This presentation, from the 2020 International Conference on Functional Programming, describes a new technique that is used to make combinator parsers very, very efficient.
- The talk begins with a great, accessible introduction, which runs for about three and a half minutes. Then around 31:44 it accelerates into hyperspace (a parsing machine based on continuations). In my opinion, the typed machine at 35:00 is very cool.
- If you want to follow along with the talk, here is a translation table:

Talk code	Our code
`pure`	`succeed`
`(:)`	`curry op ::`
`runParser`	`produce` (roughly)

Talk code	Our code
`some`	`many1`
`satisfy`	`sat`

(5) *Learn how parsing combinators work.* Read my handout on how parsing combinators work.

Lab

Lab proceeds in three parts. I will keep time and will move people along in lock step.

(6) *Combinators as functions.* From the parsing handout, review the type of `'a producer` and the three possible forms of its result type:

- Failure: `NONE`
- Success: `SOME (OK a, remaining_inputs)`
- Error: `SOME (ERROR msg, remaining_inputs)`

Now define the parsing combinator for choice:

```
val <|> : 'a producer * 'a producer -> 'a producer
```

You may define an ML function using `fun`, or you may complete this algebraic law:

```
(p1 <|> p2) ts == ...
```

Plan on writing one case for each form of result from `p1 ts`.

If you finish before I move the group to step (7), define the parsing combinator for satisfaction:

```
val sat : ('a -> bool) -> 'a producer -> 'a producer
```

Again you may write code or you may complete this law:

```
sat predicate p ts = ...
```

Again you will have to consider all possible forms of `p ts`.

(7) *Combinators as abstractions.* Although an `'a producer` is represented as an ML function, we usually treat a producer as an abstraction. In the next part of the lab, you will define higher-order functions that make new combinators from old combinators. Your code will not acknowledge that a parsing combinator is a function. Instead, you will build new combinators using operations like `<$>`, `<*>`, and `<|>`. Function `curry` will also be useful.

The first group of producers you will define are the ones that implement classic notations of extended BNF: "zero or one," "zero or more," and "one

or more." You'll also implement a notation found in Internet protocols and other odd corners: "exactly N."

```
val optional : 'a producer -> 'a option producer  (* zero or one *)
```

Remember the definition of `'a option`:

```
datatype 'a option = SOME of 'a | NONE
```

To show `optional` in action, I use it to get the optional `else` clause in a C `if` statement:

```
the "if" >> curry3 C_IF <$>
                (the "(" >> exp <~> the ")") <*>
                statement <*>
                optional (the "else" >> statement)
```

Now **write algebraic laws for `optional`**:

```
optional p == ...
```

The next two combinators recognize *lists*:

```
val many  : 'a producer -> 'a list producer       (* zero or more *)
val many1 : 'a producer -> 'a list producer       (* one or more *)
```

For example, a Scheme function application is an expression followed by zero or more arguments, all in brackets:[1]

```
the "(" >> curry APPLY <$> exp <*> many exp <~> the ")"
```

The algebraic laws for `many` and `many1` are analogous to the structure of the outputs they produce. You may recognize some laws from CS 105:

- Parser `many p` may succeed by reading no input and producing the empty list, or it may succeed by producing a single input from `p`, followed by a list of zero or more `p`'s.
- Parser `many1 p` may succeed by producing a single input from `p`, followed by a list of zero or more `p`'s.

Complete these algebraic laws:

```
many  p == ...

many1 p == ...
```

If you finish before I move the group on to step (8), try *one* of the following:

- Analogously with `many` and `many1`, define a `count` combinator that produces a list of exactly N values of type `'a`:

[1] Examples not typechecked!

```
val count : int -> 'a producer -> 'a list producer (* exactly N *)
count 0     p == ...
count (n+1) p == ...
```

- Define two more combinators that help deal with concrete syntax that must be parsed, but that does not contribute anything to a result of type `'a` in an `'a producer`. Such syntax usually includes keywords as well as syntactic particles like brackets and commas. This kind of syntax can be ignored by using the following combinators:

  ```
  val <~> : 'a producer * 'b producer -> 'a producer
  val >>  : 'a producer * 'b producer -> 'b producer
  ```

  In the first combinator, the result from the `'a producer` is used and the result from the `'b producer` is ignored; in the second combinator, the result from the `'b producer` is used and the result from the `'a producer` is ignored. Using `<$>`, `<*>`, and `curry`, among other functions, complete the following algebraic laws:

  ```
  p <~> q == ...


  p >>  q == ...
  ```

- Define a combinator

  ```
  val bracketed : 'a producer -> 'a producer
  ```

  where `bracketed p` recognizes whatever `p` recognizes, but enclosed in the matching brackets of your choice.

(8) *Writing parsers.* The last step of lab is to use what you've written, together with what I provide, to write a very simple parser.

Each token of assembly language can be recognized by a parser dedicated to that token. The parsers I provide for this lab recognize, register numbers, literals, and names. Plus `the` can recognize any single given token, like a comma or a bracket.

```
type reg = int
val reg    : reg producer             (* parses a register number *)
val int    : int producer            (* parses an integer literal *)
val string : string producer          (* parses a string literal *)
val name   : string producer          (* parses a name *)
val the    : string -> unit producer  (* one token, like a comma or bracket *)
```

As functions you can use with `<$>`, I provide these encoders:

```
type opcode = string
type instr (* instruction *)
val eR0 : opcode -> instr
val eR1 : opcode -> reg -> instr
val eR2 : opcode -> reg -> reg -> instr
```

```
val eR3 : opcode -> reg -> reg -> reg -> instr
```

The parsers and functions above are meant to be combined with `<$>`, `<*>`, `<~>`, `<|>`, and the other parsing combinators. As an example, here's a parser that recognizes the assembly-language instruction `r3 := r1 + r2`:

```
eR3 "add" <$> reg <~> the ":=" <*> reg <~> the "+" <*> reg
```

Emulating this example, write a parser for *one* of the syntactic forms you defined before lab in step (3).

After lab

Setup

(9) *Set up your computer for ML programming.* Read the handout on "Software development in Standard ML", and get set up with some ML compilers. (It sounds crazy, but I recommend that you use a mix of three ML compilers. If this sounds awful, you can limp along with just Moscow ML or MLton.)

If you're comfortable working on the department servers, you can skip this step.

(10) *Download infrastructure for universal forward translation.* 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.

(11) *Verify that the code builds.* Assuming you have Moscow ML, change to the `src/uft` directory, run `make`, then `../../bin/uft`. Running the UFT should name seven languages, of which only two are implemented so far:

```
> make
...
> ../../bin/uft
Usage:
  ../../bin/uft <from>-<to> [file]
where <from> and <to> are one of these languages:
  ho!  Higher-order vScheme with mutation
  ho   Pure, higher-order vScheme
  fo   Pure, first-order vScheme
  cl   Pure, first-order vScheme with closure and capture forms
  kn   K-Normal form
  vs   VM assembly language
  vo   VM object code
```

If you don't have Moscow ML but you do have MLton, try `make mlton` and then run `../../bin/uft.opt`.

If you haven't done so already, add the project `bin` directory to your Unix `PATH`.

(12) *Verify that the Universal Forward Translator behaves as expected.* Close study of the UFT is going to wait until we have more languages to work with. For now, confirm that you can activate two translations:

- Verify that you can use my escape-hatch parser to translate some assembly code into object code:

```
> echo '@ print 33' | uft vs-vo | svm
.load module 1
print 33
```

- Verify that you can unparse code from the escape-hatch parser, but that unparsed assembly language is just a placeholder.

```
> echo '@ print 33' | uft vs-vs
an unknown assembly-code instruction
```

If you compiled with MLton, you'll use `uft.opt` instead of `uft`.

(13) *Familiarize yourself with the big picture.* Still in the `src/uft` directory, have a quick look at these files:

`all.cm` List of all source files (except `main.sml`)

`object-code.sml` and `asm.sml` Internal representations of object code and assembly code.

`asmparse.sml` Toy parser and unparser for assembly code, including example cases for three demonstration instructions

More details can be found in the code overview.

Syntax design

(14) *Decide on concrete syntax for register names.* I ship a lexical analyzer that accepts both `$r`*k* and `r`*k* as register numbers, where *k* is an integer literal in the range 0 to 255. If that works for you, you're done. Otherwise, you'll have to edit `asmlex.sml` (the `registerNum` function, the `token` parser, or both).

(15) *Decide if you want floating-point literals.* If so, you'll need to extend `asmlex.sml`. If not, you can live nicely without them.

(16) *Finish your grammar.* In this step, which can be interleaved with with step (17), you finish designing the assembly-language syntax you started in step (3).

- Make sure there is concrete syntax for each of the five forms of assembly-language instruction defined in file `asm.sml`.
- Make sure the concrete syntax for `LOADFUNC` does not require a count of the instructions in the body (`instr list`). Instead, make sure that the concrete syntax includes some sort of unambiguous terminator, so a parser can tell where the body of the function stops. (The terminator must look different from an instruction!)
- Make sure there is concrete syntax for each of the instructions your VM recognizes (all of which will take the `OBJECT_CODE` form in `asm.sml`).
- Follow the design guidelines.

Unparsing

(17) *Write unparsers for your assembly language.* Using the grammar you defined in steps (3) and (16), and possibly concurrently with step (16), define unparsing functions so that every instruction and every assembly-language form can be rendered in the concrete syntax you have chosen.

In file `asmparse.sml`, you will find a prototype unparser `unparse1`, which is meant to unparse any instruction except `LOADFUNC`. The prototype knows how to unparse three VM instructions:

- Opcode `+` adds two registers and places the sum in a third.
- Opcode `swap` swaps the contents of two registers.
- Opcode `+imm` adds a register and an immediate constant and places the sum in a register. The immediate constant is "offset-coded"; an 8-bit byte z codes for the number $z - 128$.

The concrete syntax I use for these instructions is described in step (18).

File `asmparse.sml` also includes a prototype `unparse`, but once `LOADFUNC` enters the picture, this prototype will have to be rewritten.

Complete both steps:

A. In file `asmparse.sml`, replace prototypes `unparse` and `unparse1` with a real unparser for assembly code.
   - Write `unparse1`, which should handle every form except `LOADFUNC`. (Each of those cases produces one line of output.)
   - Write `unparse`, which handles both forms of `LOADFUNC`.[2] A function load generally requires multiple lines of assembly code to express the body of the function. *Those lines should be indented.*

[2] There is not only `Asm.LOADFUNC`, but also `Asm.OBJECT_CODE` applied to `ObjectCode.LOADFUNC`.

Function `unparse1` requires a ton of case analysis: pattern matching on opcode names. *Every instruction your VM accepts must unparse to a readable assembly-language syntax.*

B. Test your unparser by creating a test file `unparse.vs`. In this file, using my "passthrough parser", or using your own parser if you have completed step (18), write at least *half* the instructions that your VM recognizes. (If you use my passthrough parser, you will be limited to instructions in formats `R0` through `R3`, like `halt` and `print` but not `loadliteral` or `check`.) Test that these instructions unparse correctly by running

```
uft vs-vs unparse.vs
```

and comparing the output with what you expect from your grammar.

The other half of your instructions should be implemented in your unparser, but they can remain untested until next week.

One step toward parsing

(18) *Extend my placeholder parser with at least one new syntactic form.* In file `asmparse.sml`, I define a simple parser for assembly instructions. This parser, `one_line_instr`, recognizes these forms:

- From the form `@ opcode { int }`, it generates an object-code instruction with the given opcode and registers. This form basically works as a "passthrough" to be able to write some virtual object code after the `@` sign.
- From the form `reg := reg + reg`, it generates an add instruction (opcode `+`).
- From the form `reg := reg + int`, it generates an add-immediate instruction (opcode `+imm`).
- From the form `reg := reg - int`, it *also* generates an add-immediate instruction—subtracting a small integer is the same as adding a negative small integer.
- From the form `reg, reg' := reg', reg`, it generates a swap instruction (opcode `swap`). The concrete syntax is "multiple assignment," which is found in Python, Lua, and other languages.

Extend `one_line_instr` with at least one case of concrete syntax in your assembly language. A good case to implement would be concrete syntax for loading a literal value into a register. You'll add more cases next week.

(19) That's it! Next week you'll complete your parser, and you'll write the label-elimination pass that translates assembly code to virtual object code. But for now, you're done.

What and how to submit

(20) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.03`. That file needs to be edited to answer the same questions you answer every week.

From now on, **you'll be submitting your full `src` tree,** including both the UFT and the SVM.

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 hw03 src
```

(21) 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 reflection03 REFLECTION
```

Reading in depth

Occasionally I'll suggest reading that may enrich your view of programming-language implementation. This week I suggest three works on combinator parsing:

- Graham Hutton, Higher-Order Functions for Parsing (1992). The work is somewhat primitive, and the notation is obsolete, but It's a good paper. Plus Hutton was first, and in academia, we honor those who came first.
- Graham Hutton and Erik Miejer, Monadic Parser Combinators (1996). Parsing brought up to date with Haskell. Also a decent introduction to monads. A shorter version was published in the *Journal of Functional Programming* in 1998.
- Doaitse Swierstra, Combinator Parsing: A Short Tutorial (2008). Swierstra has done more than anyone else to develop the art and science of combinator parsing.

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Algebraic laws.* You can write algebraic laws for particular monads.
2. *Parsing combinators.* You can write new parsing combinators.
3. *Syntax design.* You can explain how the design of your concrete syntax promotes readability.
4. *Costs of design.* You can argue whether a well-designed concrete syntax is unreasonably expensive.
5. *Unparsing.* At least half the instructions your VM recognizes can be unparsed by your UFT.
6. *Parsing in theory.* You understand when your parser needs to see multiple tokens before making a choice.
7. *Parsing in practice.* Using a parser you wrote yourself, your UFT can successfully parse one assembly-language instruction.
8. *Embedding and projection.* You understand how assembly-language abstract syntax and concrete syntax are related by embedding and projection.

Learning outcome available for depth points:

9. *Assembler macro [1 point].* You make `check-expect` look like a single assembly instruction, but it expands to multiple VM instructions. You can assemble and run a `.vs` file that contains a `check-expect`. *The point can be redeemed with this module or the next module; after that, it expires.*

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. In lab you wrote algebraic laws for Kleisli composition using the baby error monad. To claim this point, write algebraic laws for Kleisli composition using the full error monad.
2. To claim this point, define this parsing combinator:

   ```
   val commaSep : 'a producer -> 'a list producer
     (* `commaSep p` returns a parser that parses a sequence
         of zero or more p's separated by commas *)
   ```

   You may include the definition in your `REFLECTION` file or in your Monday night submission (in which case, please point to it).

3. To claim this point, identify examples of two different syntactic forms of *virtual object code*, and compare those examples side by side with the corresponding forms of your assembly language. Explain why you believe that the assembly-language examples are more readable.

4. It's good if the assembly language is easy to read and write, but it's also good if the code inside the UFT is short. To claim this point, argue at least one of the following positions:

   - These are competing concerns.
   - Both can be accomplished in a single design.

   To earn the point, every argument must be supported by pointing to specific line numbers within your unparser.

5. To claim this point, make sure your Monday night submission includes test file `unparse.vs` and that it passes the test described in step (17)B above.

6. To claim this point, identify a choice point in your grammar (that is, an alternative between two syntactic forms), where making the choice requires looking at more than one token. If all your choices can be made by looking at a single token—that is, if your grammar is LL(1)—then explain what you did to make that so.

7. To claim this point, you will show assembly syntax that your UFT successfully parses and unparses, using code you wrote yourself (or with a partner). You may use the parser you wrote in lab in step (8), or you may use another parser of your choice. Demonstrate that it works with your unparser by feeding an assembly language instruction to the UFT, parsing it, and unparsing it back to the original string, as in this example from my own code:

   ```
   > echo "r1 := r2 + r3" | uft vs-vs
   r1 := r2 + r3
   ```

8. To claim this point,

   - Explain whether and how the embedding/projection concept applies to the abstract syntax and concrete syntax of assembly language.
   - Point to where in your `code` embedding and/or projection are implemented.
   - If projection is implemented, say how projection failure is manifested.

Concrete syntax of the week

Your most interesting design choice will probably be the notation you want to use for global variables. Here are your options:

- *A global variable is just a name.* This is a lovely choice, but it's hard to make work. What if some joker gives you a Scheme program with a global variable called `$r0`? How are you going to distinguish it from register 0?
- *A global variable is a name with a prefix.* This choice was used in C compilers and linkers for many years: the C compiler slapped an underscore in front of every global name, and that made sure that a global name wouldn't conflict with other things. You could consider some other character as a prefix or "sigil," like Perl's `@` or `%` characters. Or you could just stick a special token before the name of a global variable.
- *A global variable is a table lookup.* At run time, the SVM indexes into a table of global variables, so if you're comfortable having a global variable look like a table reference, all you need is to invent a concrete syntax for table references.

Choose wisely!