Lesson 0 - Parsing parsers

Introduction

Parsers are everywhere. Your browser has many, from rendering websites to verifying the identity of the server you're connected to. Your word processor likely supports several file formats, like *.rtf*, *.doc*, *.docx*, or even *.odt*.

Parsers have two jobs: Enforcing rules (*validation*) and performing computation. Good parsers *always enforce the rules of their protocol*, validating that the message is syntactically correct **before** calling potentially dangerous functions like `malloc` or `strcpy`.

Unfortunately, many parsers are written in a manner that makes it hard to audit; a lot of major vulnerabilities are fundamentally parser bugs. This tutorial introduces parser combinators as a style of writing parsers, and teaches the fundamentals of the Hammer parser combinator library.

Some knowledge of C is useful, especially in debugging your parser.

Whitelisting vs Blacklisting

A better parser

Parsers have a lot of moving parts, all of which have properties that must be enforced. *Parser Combinators* are a simple way of defining those rules. We use smaller parsers and combine them to form a complete parser.

This has two effects.

1. Your parser now has a clear boundary where your grammar is specified.
2. The code you write to do computation no longer has anything to do with checking syntax. It's enforced for you by default.

Installing hammer

First, you need to obtain hammer, either downloading a zip or cloning it.

Then you'll `cd` into the directory - use `scons` to build it, and `scons install` to move the library/headers to a system-wide directory. May need to run that one as sudo.

Setting up cling

Cling is really nice - Built by the wonderful people at CERN, it lets you run `C/C++` code by interpreting it as you go. You can download a binary package for Cling here. Running Cling should be as simple as extracting the archive and running `bin/cling` from the command line.

To get `Hammer` working in `Cling` I had to specify the path for Hammer's library as well as OS X's include path. On OS X 10.10:

```
./cling
$ .I /Applications/Xcode.app/Contents/Developer/
Platforms/MacOSX.platform/Developer/SDKs/
MacOSX10.10.sdk/usr/include
$ .L /usr/local/lib/libhammer.dylib
```

Now you can just include Hammer and go:

```
$ #include <hammer/hammer.h>
```

Note that when you execute a C statement in cling, it prints the return value. I'll include that in the output for the rest of this tutorial, make sure not to copy and paste that in if you follow along!

Lesson 1 - Basic combinators

In this lesson, we'll introduce a few basic building blocks. By the end you should be able to see how they can be combined into something interesting!

`h_ch()`, `h_sequence()`, `h_end_p()` are introduced.

Scenario 1

Eugene wants to build a parser that accepts friendly messages. Particularly the string "hi". Turns out this is rather simple in Hammer!

Hammer provides several basic parsers, including one that eats characters: `h_ch`. Here's an example of using `h_ch`:

```
h = h_ch('h');
i = h_ch('i');
```

One last thing I need to mention before we continue - since this is C, we need to specify types. `h` and `i` are of type `HParser *`, that is, a pointer to a `HParser`, the generic parser type Hammer provides. We can rewrite the example so that Cling interprets it properly, as such:

```
$ HParser *h = h_ch('h')
  (HParser *) 0x7fdee2421348
$ HParser *i = h_ch('i')
  (HParser *) 0x7fdee2559aa8
```

Now `h` and `i` are variables that contain pointers to parsers, one which takes an `h` and one which takes an `i`.

We can combine `h` and `i` into a single parser that accepts a sequence of the two using the `h_sequence` parser. We assign the result to `hi`.

```
$ HParser *hi = h_sequence(h, i, NULL)
  (HParser *) 0x7fdee2419098
```

Because this is C, sequences end in a NULL byte, similar to strings! We can test this parser and show that it rejects other input.

To test the parsers, we call `h_parse` which should give us a `HParseResult`.

```
HParseResult *res = h_parse(hi, (const uint8_t*)"hi",
2)
(HParseResult *) 0x7fdee29360f0
```

We can then extract the parsed elements out:

```
$ res->ast->seq->elements[0]->uint
(uint64_t) 104
$ res->ast->seq->elements[1]->uint
(uint64_t) 105
```

It parsed them out as uints, because that's what we asked for. 104 and 105 correspond to ascii 'h' and 'i'.

Now let's verify our sequence parser `hi` parses incorrect sequences properly (that is, by failing to parse it). We can reuse the res variable, and don't have to specify a type the second time around.

```
res = h_parse(hi, (const uint8_t*)"ha", 2)
(HParseResult *) nullptr
```

`nullptr`! Correct so far.

The end (or is it?)

What if we parse a longer word?

```
$ res = h_parse(hi, (const uint8_t*)"hippo", 5)
(HParseResult *) 0x7fc18b09ecf0
$ res->ast
Segmentation fault: 11
```

Uh-oh. That's not good. Eugene doesn't want hippos in his parser. How can we fix this?

Blacklisting

If we were designing a parser the **wrong** way, we might simply add a check later on that makes sure the user doesn't pass in "hippo". Or "hippos". Or "hills". Or "history". Or…

You could imagine the list of words we don't want to accept is quite long. In fact, it's much longer than the list of words we *do* want ("hi"). This example closely resembles an **anti pattern** in parser design: `sanitization`.

The quote goes something like this: "Think of everything that isn't an elephant". Except in this case it's hippos.

Whitelisting

Let's get away from blacklisting. Thinking of all the things we don't want to accept is a pretty hard exercise. It's better if we simply define what we **do** want, and reject all else.

In our case, we can do that by using the combinator `h_end_p()`. `h_end_p()` is a special parser that expects the end of input at its position.

Let's change our parser to give it a try:

```
$ HParser *just_hi = h_sequence(h, i, h_end_p(), NULL)
(HParser *) 0x7fd7a9e06cd8
```

And test it:

```
$ res = h_parse(just_hi, (const uint8_t*)"hippo", 5)
(HParseResult *) 0x0
```

Good. A NULL return value means the parse failed, as it should have.

All of those type signatures can get a little tedious - Hammer has some nice macros to help with this, like H_RULE. To use them, #include <hammer/glue.h>.

```
HParser *h = h_ch('h')
```

is equivalent to

```
H_RULE(h, h_ch('h'))
```

The payoff might not be obvious just yet, but it makes it very quick to attach further validation or semantic actions onto the parsers.

Lesson 2 - A light should go on...

`h_optional`, `h_choice`, `h_many1`

Melody likes lightbulbs, but really wants to control hers remotely. Deciding to hook it up to the internet, she does some research and sees lots of warnings about the Internet Of Things™. She certainly doesn't want to accept **just any** any evil attacker input! Whatever is Melody to do?

Naturally she picks Hammer as her parser combinator library. Next she brainstorms what her requirements are, and lists a few:

1. The lightbulb defaults to an off state.
2. Sending an "Off" message is only valid if the lightbulb is currently "On".
3. Sending an "On" message is only valid if the lightbulb is currently "Off".
4. An empty message is invalid.

5. Messages are a combination of “On”s and “Off”s, nothing else.

Melody brings up her Cling prompt and types in `#include <hammer/hammer.h>`. She adds `#include <hammer/glue.h>` as she wants to use `H_RULE` macros for brevity.

First, she binds the `on` and `off` local variables to parsers that accept the strings “On” and “Off”.

```
$ H_RULE(on, h_sequence(h_ch('O'), h_ch('n'), NULL))
$ H_RULE(off , h_sequence(h_ch('O'), h_ch('f'),
h_ch('f'), NULL))
```

She initially wants to create a basic parser that can be either `on` or `off`. She uses the `h_choice` combinator to do this:

```
H_RULE(lightswitch_message, h_choice(on, off, NULL))
```

What about multiple messages? You might want to turn the lightbulb On, then Off.

We can use the `h_many` combinator for this. `h_many` takes a parser and applies it any number of times. (`h_many1` applies a parser at least 1 time).

```
H_RULE(many_messages, h_many(lightswitch_message))
```

This isn’t quite right, though. Sure, these parsers will only accept “On” and “Off”. That covers requirement 5, but not requirements 2 and 3.

Syntax vs Semantics

It turns out we've covered the **Syntax** of our lightbulb message protocol. We only accept "On" or "Off".

When we start talking about valid "states", like whether you can turn an "On" lightbulb "On", we get into **Semantics**. It's worth knowing the difference!

Here's an example in English, composed by Noam Chomsky:

Colorless green ideas sleep furiously.

This sentence is syntactically correct because it follow the grammar rules prescribed in elementary school.

The sentence is meaningless, though. It has no **semantic** meaning.

Just like telling an "Off" lightbulb to turn "Off".

Semantic success

Melody brainstorms some more. She begins by writing some valid sequences of messages and starts noticing patterns which she marks with parenthesis.

```
 On
(On, Off,)
(On, Off,)  On
(On, Off,) (On, Off)
(On, Off,) (On, Off,)  On
(On, Off,) (On, Off,) (On, Off)
(On, Off,) (On, Off,) (On, Off,) On
```

After staring at her screen a while, she comes up with a prose explanation of the rules she needs to implement in Hammer. She writes it out:

> A message is either:
>
> 1. Just "On", Or:
> 2. Any number of matching ("On","Off") pairs, with an optional trailing "On".

After staring at her definition some more, she feels confident this covers her initial requirements. She begins transcribing English to parsers:

```
H_RULE(just_on, h_sequence(on, h_end_p(), NULL))
```

("On", followed by `end_of_input`)

```
H_RULE(on_off, h_sequence(on, off, NULL))
```

(A pair of "On" and "Off")

```
H_RULE(valid_message, h_choice(just_on,
h_sequence(h_many1(on_off), h_end_p(), NULL),
NULL));
```

This rule is nearly correct! All she's missing is an optional "On" at the end. A master of the Hammer framework, she knows she can use `h_optional` to cover just this case.

```
H_RULE(valid_message, h_choice(just_on,
```

```
h_sequence(h_many1(on_off), h_optional(on), h_end_p(),
NULL),
NULL));
```