Hammer: Smashing Binary Formats Into Bits

mlp and tq

Upstanding Hackers

July 27, 2012

The recursive-descent family

- Recursive descent parsers
  - Parsing like mom used to do it (if your mom is Jack Crenshaw)
  - Conceptually really simple
  - Can't do left recursion
  - Can have exponential runtime

The recursive-descent family

- Recursive descent parsers
  - Parsing like mom used to do it (if your mom is Jack Crenshaw)
  - Conceptually really simple
  - Can't do left recursion
  - Can have exponential runtime
- Parsing expression grammars
  - Look a lot like CFGs
  - Always deterministic (unambiguous)
  - Provide lookahead
  - Still can't do left recursion

The recursive-descent family

- Recursive descent parsers
  - Parsing like mom used to do it (if your mom is Jack Crenshaw)
  - Conceptually really simple
  - Can't do left recursion
  - Can have exponential runtime
- Parsing expression grammars
  - Look a lot like CFGs
  - Always deterministic (unambiguous)
  - Provide lookahead
  - Still can't do left recursion
- Packrat parsers
  - They're PEGs, but memoized
  - Can handle left-recursion!

Why binary parsing?

- None of the existing tools do it well
  - Limited to character streams
  - Endianness is a pain in the dick
  - So are bit-fields

Why binary parsing?

- None of the existing tools do it well
  - Limited to character streams
  - Endianness is a pain in the dick
  - So are bit-fields
- Except bison, which nobody likes
  - Interface sucks for everything except parsers/interpreters
  - Shift-reduce conflicts are confusing
  - Bit-fields still hard unless everything's nicely byte-aligned

Requirements

- Thread-safe and reentrant
- Simple API
- Fast
- Correct

Naming conventions

- Types
  - Start with **H** and are CamelCased
  - `HParser`, `HParsedToken`, etc

Naming conventions

- Types
  - Start with **H** and are CamelCased
  - `HParser`, `HParsedToken`, etc
- Functions
  - Start with **h_** and use underscores
  - `h_parse()`, `h_length_value()`, etc

Basic usage

```
#include "hammer.h"

const HParsedToken* build_my_struct(const HParseResult *p) {
  // ...
}
int main(int argc, char** argv) {
  // obtain data, and its length, from somewhere
  // Create a parser
  HParser *parser = action(...,
                           build_my_struct);
  // Parse the data
  HParseResult *result = h_parse(parser, data, length);
  // Get your struct back from the result token and use it
  do_something(result->ast->user);
  return 0;
}
```

Result types

- HParseResult
  - A tree of parsed tokens
  - The total number of bits parsed
  - A reference to the memory context for this parse

Result types

- HParseResult
  - A tree of parsed tokens
  - The total number of bits parsed
  - A reference to the memory context for this parse
- HParsedToken
  - Token type: bytes, signed/unsigned int, sequence, user-defined
  - Token (a tagged union)
  - Byte index and bit offset

Primitives

- Character and token parsers
  - `h_ch(const uint8_t c)`, `h_token(const uint8_t *str, size_t len)`
  - `h_ch_range(const uint8_t lower, const uint8_t upper)`
  - `h_not_in(const uint8_t charset, size_t length)`

Primitives

- Character and token parsers
  - `h_ch(const uint8_t c)`, `h_token(const uint8_t *str, size_t len)`
  - `h_ch_range(const uint8_t lower, const uint8_t upper)`
  - `h_not_in(const uint8_t charset, size_t length)`
- Integral parsers
  - `h_uint8()`, `h_int64()`
  - `h_bits(size_t len, bool sign)`
  - `h_int_range(const HParser *p, const int64_t lower, const int64_t upper)`

Primitives

- Character and token parsers
  - `h_ch(const uint8_t c)`, `h_token(const uint8_t *str, size_t len)`
  - `h_ch_range(const uint8_t lower, const uint8_t upper)`
  - `h_not_in(const uint8_t charset, size_t length)`
- Integral parsers
  - `h_uint8()`, `h_int64()`
  - `h_bits(size_t len, bool sign)`
  - `h_int_range(const HParser *p, const int64_t lower, const int64_t upper)`
- End-of-input
  - `h_end_p()`

Combining primitives

- Sequential and alternative
  - `h_sequence(const HParser *p, ...)`
  - `h_choice(const HParser *p, ...)`

Combining primitives

- Sequential and alternative
  - `h_sequence(const HParser *p, ...)`
  - `h_choice(const HParser *p, ...)`
- Repetition
  - `h_many(const HParser *p)`
  - `h_many1(const HParser *p)`
  - `h_repeat_n(const HParser *p, size_t len)`

Combining primitives

- Sequential and alternative
  - `h_sequence(const HParser *p, ...)`
  - `h_choice(const HParser *p, ...)`
- Repetition
  - `h_many(const HParser *p)`
  - `h_many1(const HParser *p)`
  - `h_repeat_n(const HParser *p, size_t len)`
- Optional
  - `h_optional(const HParser *p)`

Combining primitives

- Sequential and alternative
  - `h_sequence(const HParser *p, ...)`
  - `h_choice(const HParser *p, ...)`
- Repetition
  - `h_many(const HParser *p)`
  - `h_many1(const HParser *p)`
  - `h_repeat_n(const HParser *p, size_t len)`
- Optional
  - `h_optional(const HParser *p)`
- Not Actually Appearing In This Parse Tree
  - `h_ignore(const HParser *p)`

Combining primitives

- Sequential and alternative
  - `h_sequence(const HParser *p, ...)`
  - `h_choice(const HParser *p, ...)`
- Repetition
  - `h_many(const HParser *p)`
  - `h_many1(const HParser *p)`
  - `h_repeat_n(const HParser *p, size_t len)`
- Optional
  - `h_optional(const HParser *p)`
- Not Actually Appearing In This Parse Tree
  - `h_ignore(const HParser *p)`
- Higher-order
  - `h_length_value(const HParser *length, const HParser *value)`
  - `h_and(const HParser *p)`, `h_not(const HParser *p)`
  - `h_indirect(const HParser *p)`

Doing things to combinations of primitives

- `h_attr_bool(const HParser *p, const HPredicate pred)`

Doing things to combinations of primitives

- `h_attr_bool(const HParser *p, const HPredicate pred)`
- `h_action(const HParser *p, const HAction a)`

Top-level DNS

```
const HParser *dns_message =
h_action(h_attr_bool(h_sequence(dns_header,
                                h_many(dns_question),
                                h_many(dns_rr),
                                h_end_p(),
                                NULL),
                     validate_dns),
         pack_dns_struct);
```

DNS Questions

```
const HParser *dns_question =
h_sequence(h_sequence(h_many1(h_length_value(h_int_range(
                                                    h_uint8(),
                                                    1,
                                                    255),
                                                h_uint8()))),
                      h_ch('\x00'),
                      NULL),   // QNAME
           qtype,              // QTYPE
           qclass,             // QCLASS
           NULL);
```

DNS RRs

```
const HParser *dns_rr =
h_sequence(domain,                                  // NAME
           type,                                    // TYPE
           class,                                   // CLASS
           h_uint32(),                              // TTL
            // RDLENGTH+RDATA
           h_length_value(h_uint16(), h_uint8()),
           NULL);
```

Validating a DNS packet

```
bool validate_dns(HParseResult *p) {
  if (TT_SEQUENCE != p->ast->token_type)
    return false;
  HParsedToken **elems = p->ast->seq->elements[0]->seq->
                         elements;
  size_t qd = elems[8]->uint;
  size_t an = elems[9]->uint;
  size_t ns = elems[10]->uint;
  size_t ar = elems[11]->uint;
  HParsedToken *questions = p->ast->seq->elements[1];
  if (questions->seq->used != qd)
    return false;
  HParsedToken *rrs = p->ast->seq->elements[2];
  if (an+ns+ar != rrs->seq->used)
    return false;
  return true;
}
```

What's next?

- More parsing backends
  - LL(k)
  - GLR
  - LALR(8)
  - Derivatives
- Benchmarking :)
- Bindings for python, ruby, (your preferred language here)

More to come!

- Watch langsec-discuss@lists.langsec.org for further announcements
- https://github.com/UpstandingHackers/hammer