PARSING SAFELY, FROM 500MB/S TO 2GB/S

CODE MESH 2018

Geoffroy Couprie
@gcouprie

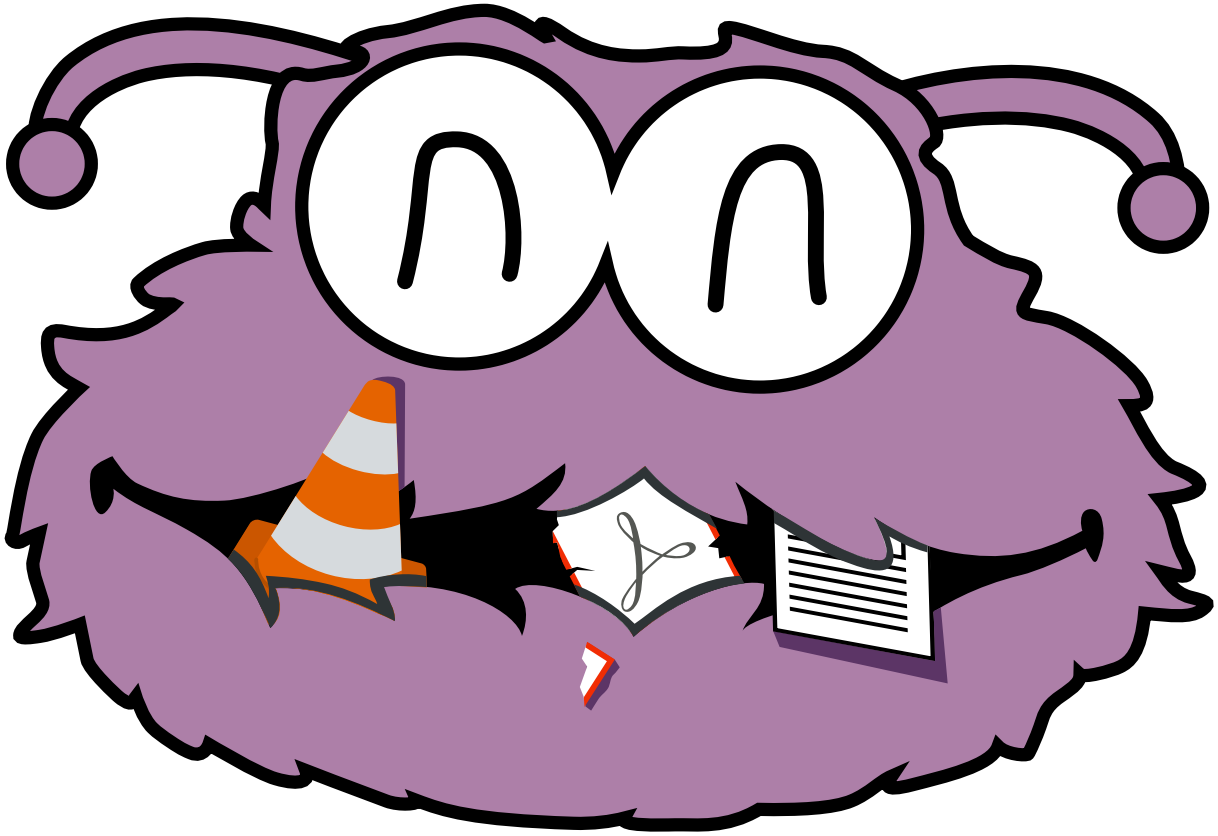
HI, I'M GEOFFROY COUPRIE

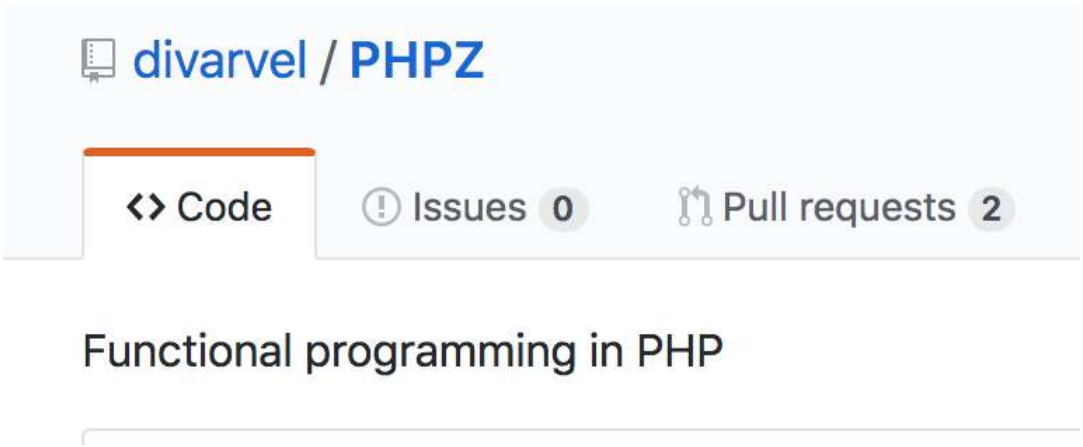
- Security and QA at Clever Cloud
- Freelance consultant, security and architecture
- works on the sozu HTTP reverse proxy
- @gcouprie

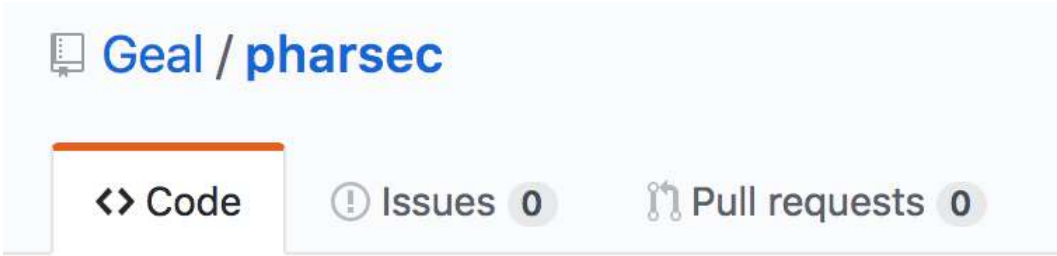

DEPARTMENT OF IT AUTOMATION
CLEVER CLOUD

NOM

fast, safe parser combinators in Rust




divarvel / PHPZ
Code
Issues 0
Pull requests 2
Functional programming in PHP


Geal / pharsec
Code
Issues 0
Pull requests 0
A parser combinator library for PHP

LOOK AT THIS DELIGHTFUL LITTLE GARBAGE LANGUAGE

```
<html>
<head></head>
<body>
<<span class="ot">?</span>
  a = <span class="st">"Hello "</span> + <span class="st">"World!"</span>
<span class="kw">?></span>
  <h1><<span class="ot">?</span> <span class="fu">print</span> a <span class="kw">?><

<<span class="ot">?</span>
    <span class="kw">A</span>  = {
    -\ value
    +/ mul = <span class="ot">(</span>x<span class="ot">,</span> y<span class="ot">)<
    -/ <span class="fu">__construct</span> = <span class="ot">(</span>val<span class=
    -/ show = <span class="ot">()</span> -> { <span class="fu">print</span> this.valu
  }

str = <span class="st">"<script>console.log('hello')</script>"</span>
obj = <span class="kw">new</span> <span class="kw">A</span><span class="ot">(</span>s
obj.show<span class="ot">()</span>
<span class="kw">?></span>
</body>
</html>
```

RUST

PARSER COMBINATORS IN 30S

- recursive descent parsing
- use "combinators" to assemble simple parsers into complex ones
- do not resolve ambiguities (have fun at runtime)

NOM PARSER INTERFACE

```
fn parser<Input, Output, Error>(i: Input)
  -> Result<(Input, Output), nom::Err<Input, Error>>;
```

NOM'S PERFORMANCE TRICK

```
fn parser<'a, Error>(i: &'a[u8])
  -> Result<(&'a[u8], &'a[u8]), nom::Err<&'a[u8], Error>>;
```

COMBINATOR DESIGN

COMBINATOR DESIGN

- first idea: each parser has its own type, assemble them through generic combinator types

COMBINATOR DESIGN

- first idea: each parser has its own type, assemble them through generic combinator types
- second idea: MACROS

(AB)USING RUST MACROS

```
named!(data, terminated!( alpha, digit ));
```

```
#[macro_export]
macro_rules! terminated(
  ($i:expr, $submac:ident!( $($args:tt)* ), $submac2:ident!( $($args2:tt)* )) => (
    {
      use $crate::lib::std::result::Result::*;

      match tuple!($i, $submac!($($args)*), $submac2!($($args2)*)) {
        Err(e) => Err(e),
        Ok((remaining, (o,_)))    => {
          Ok((remaining, o))
        }
      }
    }
  );

  ($i:expr, $submac:ident!( $($args:tt)* ), $g:expr) => (
    terminated!($i, $submac!($($args)*), call!($g));
  );

  ($i:expr, $f:expr, $submac:ident!( $($args:tt)* )) => (
    terminated!($i, call!($f), $submac!($($args)*));
  );

  ($i:expr, $f:expr, $g:expr) => (
    terminated!($i, call!($f), call!($g));
  );
);
```

GENERATED CODE

```
fn data(i: &[u8]) -> Result<(&[u8], &[u8]), nom::Err<&[u8], Error>> {
    match {
            alpha(i).and_then(|(i2, o)| {
                digit(i2).map(|(i3, o2)| (i3, (o, o2)))
            })
    } {
        Err(e) => Err(e),
        Ok((remaining, (o, _))) => Ok((remaining, o)),
    }
}
```

BUILDING A FAST HTTP PARSER

- we needed a streaming parser for the sozu HTTP reverse proxy
- good testbed for ideas
- protocol full of edge cases

```
struct Request<'a> {
    method:  &'a [u8],
    uri:     &'a [u8],
    version: &'a [u8],
}

struct Header<'a> {
    name:  &'a [u8],
    value: Vec<&'a [
}

fn request<'a>(input: &<
    -> IResult<&'a[u8], (Request<<spa

  terminated!(input,
    pair!(request_line, many1!(messag
    line_ending
  )
}
```

```
<span class="kw">fn</span> request_line<<span class="ot">'a</span>>(input: &<span cla
  <span class="pp">do_parse!</span>(input,
    method: <span class="pp">take_while1!</span>(is_token)     >>
            <span class="pp">take_while1!</span>(is_space)     >>
    uri:    <span class="pp">take_while1!</span>(is_url_token) >>
            <span class="pp">take_while1!</span>(is_space)     >>
    version: http_version              >>
    line_ending                        >>
    ( Request { method, uri, version } )
  )
}

<span class="pp">named!</span>(http_version, <span class="pp">preceded!</span>(
    <span class="pp">tag!</span>(<span class="st">"HTTP/"</span>),
    <span class="pp">take_while1!</span>(is_version)
));
```

```
fn message_header<'a>(input: &
  do_parse!(input,
    name:  take_while1!(is_token)       >>
           char!(':')
    value: many1!(message_header_value) >>

    ( Header { name, value } )
  )
}

named!(message_header_value, delimited!(
    take_while1!(is_horizontal_space),
    take_while1!(not_line_ending),
    line_ending
));
```

BENCHMARK DATA

```
GET /wp-content/uploads/2010/03/hello-kitty-darth-vader-pink.jpg HTTP/1.1\r\n\
Host: www.kittyhell.com\r\n\
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; ja-JP-mac; rv:1.9.2.3) Ge
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n\
Accept-Language: ja,en-us;q=0.7,en;q=0.3\r\n\
Accept-Encoding: gzip,deflate\r\n\
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7\r\n\
Keep-Alive: 115\r\n\
Connection: keep-alive\r\n\
Cookie: wp_ozh_wsa_visits=2; wp_ozh_wsa_visit_lasttime=xxxxxxxxxx; __utma=xxxxxxxxx.x
```

I'M SURE IT'S FAST

	one_test	httparse_example_test
nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s

HEY, LETS COMPARE IT TO OTHER PRODUCTION PARSERS!

	one_test	**httparse_example_test**
nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s
httparse	211 ns/iter (+/- 5) = 1379 MB/s	463 ns/iter (+/- 15) = 1518 MB/s
picohttpparser	155 ns/iter (+/- 6) = **1877 MB/s**	326 ns/iter (+/- 20) = **2156 MB/s**

WTF?

https://github.com/Geal/nom/blob/master/README.md

nom was designed to properly parse binary formats from the beginning. Compared to the usual handwritten C parsers, nom parsers are just as fast, free from buffer overflow vulnerabilities, and handle common patterns for you:

HOW CAN WE MAKE IT FASTER?

maybe the nom parser was a bit naive

HTTPARSE

```
<span class="kw">fn</span> parse(buffer: &[<span class="dt">u8</span>]) {
  <span class="kw">let</span> <span class="kw">mut</span> headers = [httparse::Heade
  <span class="kw">let</span> <span class="kw">mut</span> req = httparse::Request::ne
  <span class="pp">assert_eq!</span>(req.parse(buffer).unwrap(), httparse::Status::Co
}
```

PICOHTTPPARSER

```
  #[repr(C)]
  #[derive(Clone)]
  struct Header<'a>(&'a
  
  #[repr(C)]
  struct Headers<'a>(&'a
  let method = [0i8; 
  let path = [0i8; 
  let mut minor_version = 
  let mut h = [Header(&[], &[]); 
  let mut h_len = h.len();
  let headers = Headers(&mut h);
  let prev_buf_len = 0;

  let ret = unsafe {
    pico::ffi::phr_parse_request(
      buffer.as_ptr() as *const _, bu
      &mut method.as_ptr(), &mut 
      &mut path.as_ptr(), &mut 
      &mut minor_version,
      mem::transmute::<*mut Header, *mut 
      &mut h_len as *
      prev_buf_len
    )
  };
  assert_eq!(ret, buffer.len() as 
}
```

FIRST IDEA: LET'S REDUCE ALLOCATIONS

BEFORE

```
<span class="kw">struct</span> Request<<span class="ot">'a</span>> {
    method:  &<span class="ot">'a</span> [<span class="dt">u8</span>],
    uri:     &<span class="ot">'a</span> [<span class="dt">u8</span>],
    version: &<span class="ot">'a</span> [<span class="dt">u8</span>],
}

<span class="kw">struct</span> Header<<span class="ot">'a</span>> {
    name:  &<span class="ot">'a</span> [<span class="dt">u8</span>],
    value: <span class="dt">Vec</span><&<span class="ot">'a</span> [<span class="dt">
}

<span class="kw">fn</span> request<<span class="ot">'a</span>>(input: &<span class="
    -> IResult<&<span class="ot">'a</span>[<span class="dt">u8</span>], (Request<<spa

  <span class="pp">terminated!</span>(input,
    <span class="pp">pair!</span>(request_line, <span class="pp">many1!</span>(messa
    line_ending
  )
}
```

AFTER

```
struct Request<'a> {
    method:  &'a [u8],
    uri:     &'a [u8],
    version: u8,
}

struct Header<'a> {
    name:  &'a [u8],
    value: &'a [u8],
}

fn request<'a,'r
    -> IResult<&'a[u8], ()> {

  do_parse!(input,
    apply!(request_line, req) >>
    apply!(headers_iter, headers) >>
    line_ending >>
    (())
  )
}
```

AFTER REDUCING ALLOCATIONS

```
<span class="kw">fn</span> request_line<<span class="ot">'a</span>,<span class="ot">'
  <span class="pp">do_parse!</span>(input,
    method: <span class="pp">take_while1!</span>(is_token)     >>
            <span class="pp">take_while1!</span>(is_space)     >>
    uri:    <span class="pp">take_while1!</span>(is_url_token) >>
            <span class="pp">take_while1!</span>(is_space)     >>
    version: http_version              >>
    line_ending                        >>
    ({
      req.method  = method;
      req.uri     = uri;
      req.version = version;
    })
  )
}
```

AFTER REDUCING ALLOCATIONS

```
fn header<'a,'h
  do_parse!(input,
    name:  take_while1!(is_token) >>
           char!(':')             >>
    value: header_value)          >>

    ({
      header.name = name;
      header.value = value;
    })
  )
}

fn headers_iter<'a,'
  let mut iter = headers.iter_mut();
  let mut input = input;

  loop {
    let h = match iter.next() {
      Some(header) => header,
      None => break
    };

    match header(input, h) {
      Ok((i, _)) => input = i,
      Err(nom::Err::Error(_)) => <spa
      e => return e,
    }
  }

  Ok((input, ()))
}
```

BENCHMARK RESULTS

	one_test	**httparse_example_test**
original nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s
no allocations	479 ns/iter (+/- 42) = 607 MB/s	881 ns/iter (+/- 30) = 797 MB/s
httparse	211 ns/iter (+/- 5) = 1379 MB/s	463 ns/iter (+/- 15) = 1518 MB/s
picohttpparser	155 ns/iter (+/- 6) = **1877 MB/s**	326 ns/iter (+/- 20) = **2156 MB/s**

1.5x to 1.7x faster, nice, but still not there

LET'S TAKE A LOOK AT JOYENT'S HTTP-PARSER

https://github.com/nodejs/http-parser/blob/master/http_parser.c

TOKEN TABLES

```
<span class="dt">static</span> <span class="dt">const</span> <span class="dt">char</span>[
/*   0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack
       0,       0,       0,       0,       0,       0,       0,
/*   8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so
       0,       0,       0,       0,       0,       0,       0,
/*  16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn
       0,       0,       0,       0,       0,       0,       0,
/*  24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs
       0,       0,       0,       0,       0,       0,       0,
/*  32 sp    33  !    34  "    35  #    36  $    37  %    38  &
       ' ',     '!',       0,       '#',     '$',     '%',
/*  40  (    41  )    42  *    43  +    44  ,    45  -    46  .
       0,       0,       0,       0,       0,     '*',     '+'
/*  48  0    49  1    50  2    51  3    52  4    53  5    54  6
      '0',     '1',     '2',     '3',     '4',     '5',     '6'
/*  56  8    57  9    58  :    59  ;    60  <    61  =    62  >
      '8',     '9',       0,       0,       0,       0,       0,
/*  64  @    65  A    66  B    67  C    68  D    69  E    70  F
       0,     'a',     'b',     'c',     'd',     'e',     'f',
/*  72  H    73  I    74  J    75  K    76  L    77  M    78  N
      'h',     'i',     'j',     'k',     'l',     'm',     'n',
/*  80  P    81  Q    82  R    83  S    84  T    85  U    86  V
      'p',     'q',     'r',     's',     't',     'u',     'v',
/*  88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^
      'x',     'y',     'z',       0,       0,       0,       0,
/*  96  `    97  a    98  b    99  c   100  d   101  e   102  f
      '`',     'a',     'b',     'c',     'd',     'e',     'f',
/* 104  h   105  i   106  j   107  k   108  l   109  m   110  n
      'h',     'i',     'j',     'k',     'l',     'm',     'n',
/* 112  p   113  q   114  r   115  s   116  t   117  u   118  v
      'p',     'q',     'r',     's',     't',     'u',     'v',
/* 120  x   121  y   122  z   123  {   124  |   125  }   126  ~
      'x',     'y',     'z',       0,     '|',       0,
```

SWITCH BASED STATE MACHINES

```
<span class="kw">enum</span> state
  { s_dead = <span class="dv">1</span> <span class="co">/* important that this is > (

  , s_start_req_or_res
  , s_res_or_resp_H
  , s_start_res
  , s_res_H
  , s_res_HT
  , s_res_HTT
  , s_res_HTTP
  , s_res_http_major
  , s_res_http_dot
  , s_res_http_minor
  , s_res_http_end
  , s_res_first_status_code
  , s_res_status_code
  , s_res_status_start
  , s_res_status
  , s_res_line_almost_done

  , s_start_req

  , s_req_method
  , s_req_spaces_before_url
  , s_req_schema
  , s_req_schema_slash
  , s_req_schema_slash_slash
  , s_req_server_start
  , s_req_server
  , s_req_server_with_at
  , s_req_path

[etc]
```

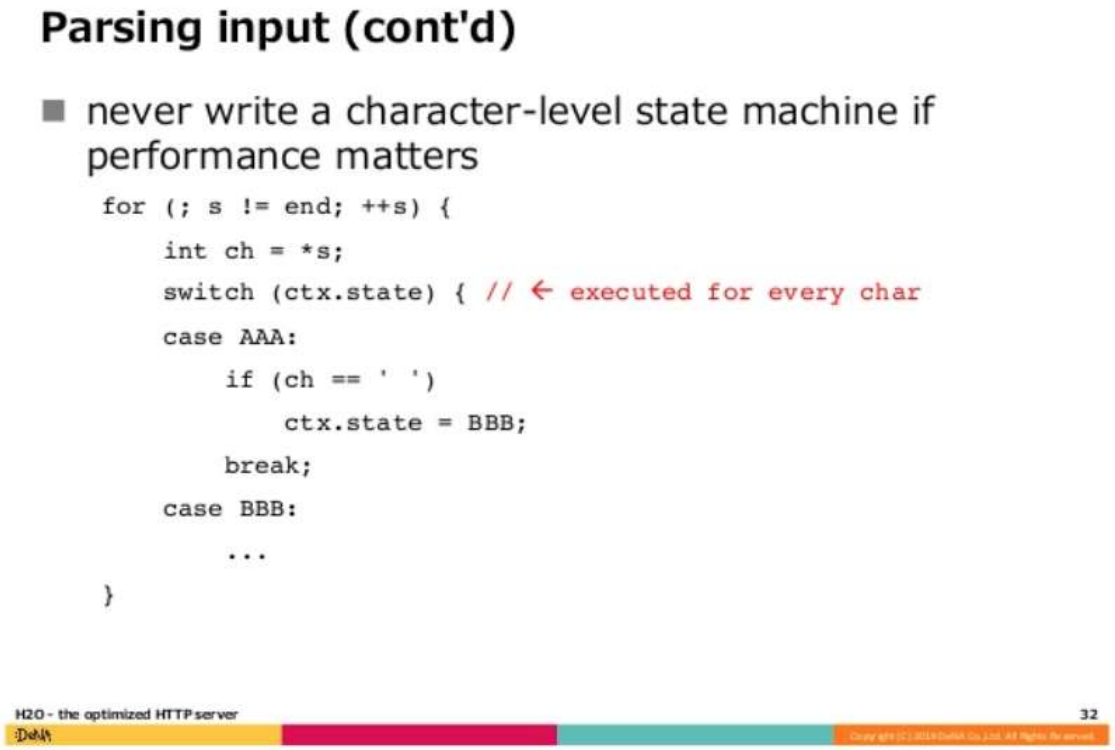
SWITCH BASED STATE MACHINES

```
  <span class="kw">for</span> (p=data; p != data + len; p++) {
    ch = *p;

    <span class="kw">if</span> (PARSING_HEADER(CURRENT_STATE()))
      COUNT_HEADER_SIZE(<span class="dv">1</span>);

reexecute:
    <span class="kw">switch</span> (CURRENT_STATE()) {

      <span class="kw">case</span> s_dead:
        <span class="co">/* this state is used after a 'Connection: close' message</s
<span class="co">         * the parser will error out if it reads another message</sp
<span class="co">         */</span>
        <span class="kw">if</span> (LIKELY(ch == CR || ch == LF))
          <span class="kw">break</span>;

        SET_ERRNO(HPE_CLOSED_CONNECTION);
        <span class="kw">goto</span> error;

      <span class="kw">case</span> s_start_req_or_res:
[etc]
```

SWITCH BASED STATE MACHINES

```
<span class="kw">case</span> s_res_H:
        STRICT_CHECK(ch != 'T');
        UPDATE_STATE(s_res_HT);
        <span class="kw">break</span>;

      <span class="kw">case</span> s_res_HT:
        STRICT_CHECK(ch != 'T');
        UPDATE_STATE(s_res_HTT);
        <span class="kw">break</span>;

      <span class="kw">case</span> s_res_HTT:
        STRICT_CHECK(ch != 'P');
        UPDATE_STATE(s_res_HTTP);
        <span class="kw">break</span>;

      <span class="kw">case</span> s_res_HTTP:
        STRICT_CHECK(ch != '/');
        UPDATE_STATE(s_res_http_major);
        <span class="kw">break</span>;

      <span class="kw">case</span> s_res_http_major:
        <span class="kw">if</span> (UNLIKELY(!IS_NUM(ch))) {
          SET_ERRNO(HPE_INVALID_VERSION);
          <span class="kw">goto</span> error;
        }

        parser->http_major = ch - '<span class="dv">0</span>';
        UPDATE_STATE(s_res_http_dot);
        <span class="kw">break</span>;
```

LET'S NOT CHANGE STATE ON EVERY CHARACTER



Parsing input (cont'd)

- never write a character-level state machine if performance matters

```
for (; s != end; ++s) {
    int ch = *s;
    switch (ctx.state) { // ← executed for every char
    case AAA:
        if (ch == ' ')
            ctx.state = BBB;
        break;
    case BBB:
        ...
}
```

H2O - the optimized HTTP server 32

(from "The internals H2O (or how to write a fast server)" by Kazuho Oku)

SECOND IDEA: LET'S IMPLEMENT THE TOKEN TABLES

current nom parser:

```
<span class="co">// used with take_while1 and other combinators</span>
<span class="kw">fn</span> is_token(c: <span class="dt">u8</span>) -> <span class="dt
    <span class="kw">match</span> c {
        <span class="dv">128.</span>..<span class="dv">255</span> => <span class="cn'
        <span class="dv">0.</span>..<span class="dv">31</span>    => <span class="cn'
        b<span class="ch">'('</span>      => <span class="cn">false</span>,
        b<span class="ch">')'</span>      => <span class="cn">false</span>,
        b<span class="ch">'<'</span>      => <span class="cn">false</span>,
        b<span class="ch">'>'</span>      => <span class="cn">false</span>,
        b<span class="ch">'@'</span>      => <span class="cn">false</span>,
        b<span class="ch">','</span>      => <span class="cn">false</span>,
        b<span class="ch">';'</span>      => <span class="cn">false</span>,
        b<span class="ch">':'</span>      => <span class="cn">false</span>,
        b<span class="ch">'</span><span class="sc">\\</span><span class="ch">'</span>
        b<span class="ch">'"'</span>      => <span class="cn">false</span>,
        b<span class="ch">'/'</span>      => <span class="cn">false</span>,
        b<span class="ch">'['</span>      => <span class="cn">false</span>,
        b<span class="ch">']'</span>      => <span class="cn">false</span>,
        b<span class="ch">'?'</span>      => <span class="cn">false</span>,
        b<span class="ch">'='</span>      => <span class="cn">false</span>,
        b<span class="ch">'{'</span>      => <span class="cn">false</span>,
        b<span class="ch">'}'</span>      => <span class="cn">false</span>,
        b<span class="ch">' '</span>      => <span class="cn">false</span>,
        _         => <span class="cn">true</span>,
    }
}
```

let's replace with the httparse version:

```
#[inline]
fn is_header_name_token(b: u8) -> bool {
    HEADER_NAME_MAP[b as usize]
}

static HEADER_NAME_MAP: [bool; 256] = byte_map![
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
];
```

BENCHMARK RESULTS

	one_test	**httparse_example_test**
original nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s
no allocations	479 ns/iter (+/- 42) = 607 MB/s	881 ns/iter (+/- 30) = 797 MB/s
token tables	285 ns/iter (+/- 10) = 1021 MB/s	590 ns/iter (+/- 58) = 1191 MB/s
httparse	211 ns/iter (+/- 5) = 1379 MB/s	463 ns/iter (+/- 15) = 1518 MB/s
picohttpparser	155 ns/iter (+/- 6) = **1877 MB/s**	326 ns/iter (+/- 20) = **2156 MB/s**

it's getting better

SIDE NOTE: CONST FN

the token table trick is nice, but writing a huge table manually is not nice

idea:

- rust has (had) const fn, ie functions that could be evaluated at compile time
- Write a function like we would do naively
- use that function to build the table at compile times

```
macro_rules! cl_map {
  ($cl: expr, $($flag:expr,)*) => ([
      $($cl($flag),)*
  ])
}
macro_rules! make_map (
  ($name: ident, $cl: expr) => (
    pub fn $name(c: dt
      static MAP: [bool;
        0x00,0x01,0x
        0x10,0x11,0x
        0x20,0x21,0x
        0x30,0x31,0x
        0x40,0x41,0x
        0x50,0x51,0x
        0x60,0x61,0x
        0x70,0x71,0x
        0x80,0x81,0x
        0x90,0x91,0x
        0xA0,0xA1,0x
        0xB0,0xB1,0x
        0xC0,0xC1,0x
        0xD0,0xD1,0x
        0xE0,0xE1,0x
        0xF0,0xF1,0x
      );
      MAP[c as usize]
    }
));

const fn is_string_token_cst(c: spar
  c != b'"' && c != b'\'' && c
}

make_map!(is_string_token, is_string_token_cst);
```

THIRD IDEA: UNROLL LOOPS

old technique, but still effective

current nom code:

```
<span class="co">//take_while1 uses the split_at_position1 function</span>
<span class="kw">fn</span> split_at_position1<P>(&<span class="kw">self</span>, predi
    -> IResult<<span class="kw">Self</span>, <span class="kw">Self</span>, <span clas
<span class="kw">where</span>
  P: <span class="bu">Fn</span>(<span class="kw">Self</span>::Item) -> <span class="c
{
  <span class="kw">match</span> (<span class="dv">0.</span>.<span class="kw">self</sp
    <span class="cn">Some</span>(<span class="dv">0</span>) => <span class="cn">Err</
    <span class="cn">Some</span>(i) => <span class="cn">Ok</span>((&<span class="kw">
    <span class="cn">None</span> => <span class="cn">Err</span>(<span class="cn">Err<
  }
}
```

LET'S WRITE A NEW COMBINATOR CALLED "TAKE_WHILE1_UNROLLED"

- go through chunks of 8 bytes of input
- apply the same operation on each byte
- keep a case where we loop manually for the last chunk

The code is almost the same!

```
<span class="pp">named!</span>(header_value, <span class="pp">delimited!</span>(
    <span class="pp">take_while1!</span>(is_horizontal_space),
    <span class="pp">take_while1_unrolled!</span>(is_header_value_token),
    line_ending
));
```

```
let mut i = 0u
let len = input.len();

loop {
  if len - i < 8 {
    break;
  }

  if !$predicate(unsafe { *input.get_
    break;
  }
  i = i+1;

  [**6 other times**]
}

if len - i < 8 {
  loop {
    if !$predicate(unsafe { *input.ge
      break;
    }
    i = i+1;

    if i == len {
      break;
    }
  }
}

if i == 0 {
  Err(Err::Error(Context::Code(input,
} else if i == len {
  Err(Err::Incomplete(Needed::Unknow
} else {
  let (prefix, suffix) = input.split_at(i);
  Ok((suffix, prefix))
}
```

BENCHMARK RESULTS

	one_test	**httparse_example_test**
original nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s
no allocations	479 ns/iter (+/- 42) = 607 MB/s	881 ns/iter (+/- 30) = 797 MB/s
token tables	285 ns/iter (+/- 10) = 1021 MB/s	590 ns/iter (+/- 58) = 1191 MB/s
unrolled loops	221 ns/iter (+/- 47) = 1316 MB/s	449 ns/iter (+/- 53) = 1565 MB/s
httparse	211 ns/iter (+/- 5) = 1379 MB/s	463 ns/iter (+/- 15) = 1518 MB/s
picohttpparser	155 ns/iter (+/- 6) = **1877 MB/s**	326 ns/iter (+/- 20) = **2156 MB/s**

we're now at the level of httparse

THE LAST STEP: SIMD

the idea of simd is to apply calculations on multiple bytes in one instruction

more specifically: use the pcmpestri instruction to get the index of the first byte in our 16 bytes chunks that is not in a range we specified

if we're left with less than 16 bytes, apply the basic loop

target specific code:

```
<span class="kw">const</span> header_value_range:&[<span class="dt">u8</span>] = b<sp

<span class="at">#[</span>cfg<span class="at">(</span>all<span class="at">(</span>any
      target_feature <span class="at">=</span> <span class="st">"sse2"</span><span cl
<span class="pp">named!</span>(header_value, <span class="pp">delimited!</span>(
    <span class="pp">take_while1!</span>(is_horizontal_space),
    <span class="pp">take_while1_simd!</span>(is_header_value_token, header_value_ran
    line_ending
));

<span class="at">#[</span>cfg<span class="at">(</span>not<span class="at">(</span>all
      target_feature <span class="at">=</span> <span class="st">"sse2"</span><span cl
<span class="pp">named!</span>(header_value, <span class="pp">delimited!</span>(
    <span class="pp">take_while1!</span>(is_horizontal_space),
    <span class="pp">take_while1_unrolled!</span>(is_header_value_token),
    line_ending
));
```

the `take_while1_simd` combinator:

```
macro_rules! take_while1_simd (
  ($input:expr, $predicate:expr, $ranges:expr) => ({
      let input = $input;
      let mut start = input.as_ptr()
      let mut i = input.as_ptr() <spa
      let mut left = input.len();
      let mut found = <span class="cr

      if left >= 16 {

        let ranges16 = unsafe { _mm_l
        let ranges_len = $ranges.len() as</s
        loop {
          let sl = unsafe { _mm_loadu

          let idx = unsafe {
            _mm_cmpestri(
              ranges16, ranges_len,
              sl, 16,
              _SIDD_LEAST_SIGNIFICANT | _SIDD_CMP_RANGES | _SIDD_UBYTE_OPS)
          };

          if idx != 16 {
            i += idx as usize;
            found = true;
            break;
          }

          i += 16;
          left -= 16;

          if left < 16 {
            break;
          }
        }
      }

      let mut i = i - start;
```

```
      <span class="kw">if</span> !found {
        <span class="kw">loop</span> {
          <span class="kw">if</span> !$predicate(<span class="kw">unsafe</span> { *ir
            <span class="kw">break</span>;
          }
          i = i+<span class="dv">1</span>;
          <span class="kw">if</span> i == input.len() {
            <span class="kw">break</span>;
          }
        }
      }

      <span class="kw">if</span> i == <span class="dv">0</span> {
        <span class="cn">Err</span>(<span class="cn">Err</span>::Error(Context::Code(
      } <span class="kw">else</span> <span class="kw">if</span> i == input.len() {
        <span class="cn">Err</span>(<span class="cn">Err</span>::Incomplete(Needed::U
      } <span class="kw">else</span> {
        <span class="kw">let</span> (prefix, suffix) = input.split_at(i);
        <span class="cn">Ok</span>((suffix, prefix))
      }
  })
);
```

BENCHMARK RESULTS

	one_test	**httparse_example_test**
original nom	824 ns/iter (+/- 25) = 353 MB/s	1,378 ns/iter (+/- 54) = 510 MB/s
no allocations	479 ns/iter (+/- 42) = 607 MB/s	881 ns/iter (+/- 30) = 797 MB/s
token tables	285 ns/iter (+/- 10) = 1021 MB/s	590 ns/iter (+/- 58) = 1191 MB/s
unrolled loops	221 ns/iter (+/- 47) = 1316 MB/s	449 ns/iter (+/- 53) = 1565 MB/s
SIMD	173 ns/iter (+/- 14) = 1682 MB/s	319 ns/iter (+/- 39) = **2203 MB/s**
httparse	211 ns/iter (+/- 5) = 1379 MB/s	463 ns/iter (+/- 15) = 1518 MB/s
picohttpparser	155 ns/iter (+/- 6) = **1877 MB/s**	326 ns/iter (+/- 20) = 2156 MB/s

ALTERNATIVE BENCHMARK: LOC

- httparse: ~750 lines
- picohttpparser: ~625 lines
- nom (original parser): ~90 lines
- nom (unrolled): ~150 lines (+ 100 lines of take_while1_unrolled)
- nom (SIMD): ~110 lines (+70 lines of take_while_simd)

SUMMING IT UP

- nom can provide adequate performance out of the box
- but be weary of allocations
- we can get more advanced techniques while keeping readability
- instead of handwritten code, we have reusable combinators

USEFUL LINKS

- https://github.com/geal/nom
- https://github.com/seanmonstar/httparse
- https://github.com/h2o/picohttpparser/
- https://github.com/nodejs/http-parser
- http://blog.kazuhooku.com/2014/11/the-internals-h2o-or-how-to-write-fast.html
- https://blog.cloudflare.com/improving-picohttpparser-further-with-avx2/

THANKS!