Achieving Parsing Sanity

with Neotoma

Sean Cribbs

Prime Motif

Achieving Parsing Sanity

with Neotoma

Sean Cribbs

Basho Technologies

sean@basho.com

Erlang FACTORY

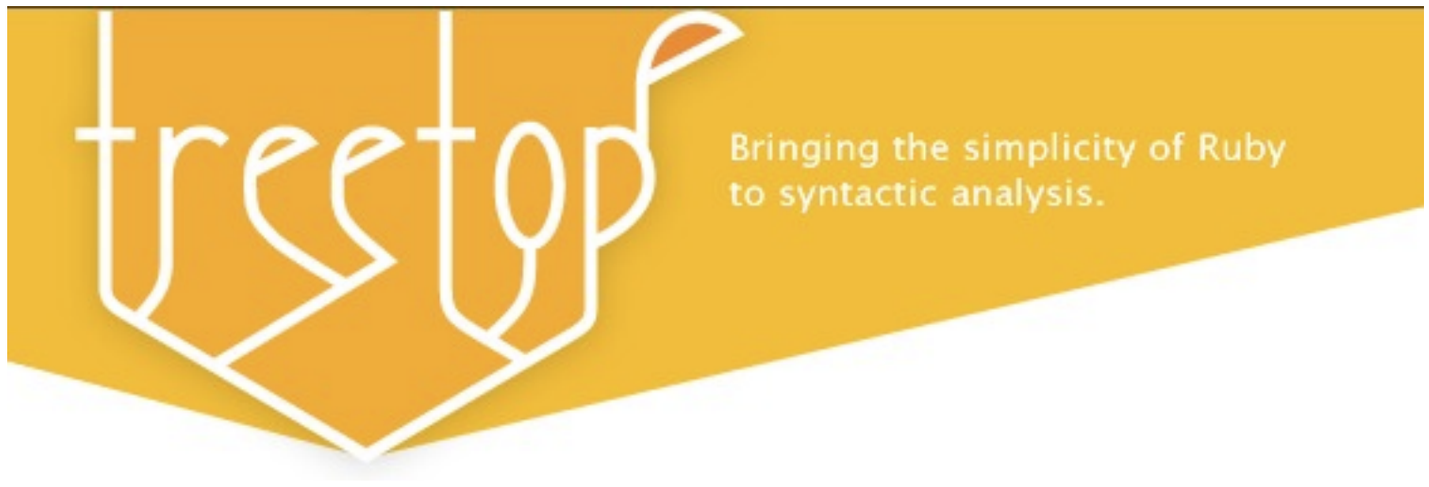
building bridges

Cucumber

```
Feature: Accepting invitations
  In order to open my account
  As a teacher or staff member
  I should be able to accept an email invitation

  Scenario Outline: Accept invitation
    Given I have received an email invitation at "joe@school.edu"
    When I follow the invitation link in the email
    And I complete my details as a <type>
    Then I should be logged in
    And the invitation should be accepted

    Examples:
      | type    |
      | staff   |
      | teacher |

  Scenario: Bad invitation code
    Given I have received an email invitation at "joe@school.edu"
    When I use a bad invitation code
    Then I should be notified that the invitation code is invalid
```

First, one must parse

Gherkin uses Treetop



OK, I'll convert to leex and yecc

```
Definitions.

D = [0-9]
IDENT = [a-z|A-Z|0-9|_|-]

Rules.

_               :  {token, {underscore, TokenLine, TokenChars}}.
\-              :  {token, {dash, TokenLine, TokenChars}}.
\%              :  {token, {tag_start, TokenLine, TokenChars}}.
\.              :  {token, {class_start, TokenLine, TokenChars}}.
#               :  {token, {id_start, TokenLine, TokenChars}}.
{D}+            :  {token, {number, TokenLine, list_to_integer(TokenChars)}}.
'(\\\^.|\\.|[^'])*' :
  S = lists:sublist(TokenChars, 2, TokenLen - 2),
  {token, {string, TokenLine, S}}.
{IDENT}+        :  {token, {chr, TokenLine, TokenChars}}.
{               :  {token, {lcurly, TokenLine, TokenChars}}.
}               :  {token, {rcurly, TokenLine, TokenChars}}.
\[              :  {token, {lbrace, TokenLine, TokenChars}}.
\]              :  {token, {rbrace, TokenLine, TokenChars}}.
\@              :  {token, {at, TokenLine, TokenChars}}.
\,              :  {token, {comma, TokenLine, TokenChars}}.
'               :  {token, {quote, TokenLine, TokenChars}}.
\:              :  {token, {colon, TokenLine, TokenChars}}.
\/              :  {token, {slash, TokenLine, TokenChars}}.
!               :  {token, {bang, TokenLine, TokenChars}}.
\(              :  {token, {lparen, TokenLine, TokenChars}}.
\)              :  {token, {rparen, TokenLine, TokenChars}}.
|               :  {token, {pipe, TokenLine, TokenChars}}.
<               :  {token, {lt, TokenLine, TokenChars}}.
>               :  {token, {gt, TokenLine, TokenChars}}.
\s+             :  {token, {space, TokenLine, TokenChars}}.

Erlang code.
```

```
Rootsymbol template_stmt.

template_stmt -> doctype : '$1'.
template_stmt -> var_ref : '$1'.
template_stmt -> iter : '$1'.
template_stmt -> fun_call : '$1'.
template_stmt -> tag_decl : '$1'.

%% doctype selector
doctype -> bang bang bang : {doctype, "Transitional", []}.
doctype -> bang bang bang space : {doctype, "Transitional", []}.
doctype -> bang bang bang space doctype_name : {doctype, '$5', []}.
doctype -> bang bang bang space doctype_name space doctype_name : {doctype, '$5', '$7'}.

doctype_name -> doctype_name_elem doctype_name : '$1' ++ '$2'.
doctype_name -> doctype_name_elem : '$1'.

doctype_name_elem -> chr : unwrap('$1').
doctype_name_elem -> dash : "-".
doctype_name_elem -> class_start : ".".
doctype_name_elem -> number : number_to_list('$1').
```

There Be Dragons Grammar EXPLODES

```
doctype -> bang bang bang.
doctype -> bang bang bang space.
doctype -> bang bang bang space doctype_name.
doctype -> bang bang bang space doctype_name space doctype_name.
```

tokens, blah

```
doctype -> bang bang bang.
doctype -> bang bang bang space.
doctype -> bang bang bang space doctype_name.
doctype -> bang bang bang space doctype_name space doctype_name.
```

tokens, blah

```
doctype -> bang bang bang.
doctype -> bang bang bang space.
doctype -> bang bang bang space doctype_name.
doctype -> bang bang bang space doctype_name space doctype_name.
```

excessively explicit

Context-Free =

Out of Context

New Project!

PEG parser-generator

Parsing Expression Grammars

- Brian Ford 2002 Thesis and related papers
- Direct representation of parsing functions, like TDPL
- Superset of Regular Expressions - terminals inline
- Ordered Choice removes ambiguities

Dangling Else Problem

if A then if B then C **else** D

if A **then** if B then C **else** D

if A then **if** B **then** C **else** D

Parsing Expressions are Functions

PE → Function

e f

sequence(e, f)

PE → Function

$e \,/\, f$

choice(e, f)

PE → Function

$$(e)$$

$$e$$

PE → Function

e+

one_or_more(e)

PE → Function

e^*

zero_or_more(e)

PE → Function

$$!e$$

not(e)

PE → Function

&e

assert(e)

PE → Function

e?

optional(e)

PE → Function

tag:e

label(“tag”, e)

PE → Function

“some text”

string(“some text”)

PE → Function

[A-Z]

character_class(“[A-Z]”)

PE → Function

.

anything()

PE → Function

*“start” e f**

sequence(string(“start”),
e, zero_or_more(f))

PE → Function

*“start” e f**

```
p_seq([p_string(“start”),
       fun e/2,
       p_zero_or_more(fun f/2)])
```

```
yeccpars0(Tokens, Tzr, State, States, Vstack) ->
    try yeccpars1(Tokens, Tzr, State, States, Vstack)
    catch
        error: Error ->
            Stacktrace = erlang:get_stacktrace(),
            try yecc_error_type(Error, Stacktrace) of
                {syntax_error, Token} ->
                    yeccerror(Token);
                {missing_in_goto_table=Tag, Symbol, State} ->
                    Desc = {Symbol, State, Tag},
                    erlang:raise(error, {yecc_bug, ?CODE_VERSION, Desc},
                                 Stacktrace)
            catch _:_ -> erlang:raise(error, Error, Stacktrace)
            end;
        %% Probably thrown from return_error/2:
        throw: {error, {_Line, ?MODULE, _M}} = Error ->
            Error
    end.

yecc_error_type(function_clause, [{?MODULE,F,[State,_,_,_,Token,_,_]} | _]) ->
    case atom_to_list(F) of
        "yeccpars2" ++ _ ->
            {syntax_error, Token};
        "yeccgoto_" ++ SymbolL ->
            {ok,[{atom,_,Symbol}],_} = erl_scan:string(SymbolL),
            {missing_in_goto_table, Symbol, State}
    end.

yeccpars1([Token | Tokens], Tzr, State, States, Vstack) ->
    yeccpars2(State, element(1, Token), States, Vstack, Token, Tokens, Tzr);
yeccpars1([], {{F, A},_Line}, State, States, Vstack) ->
    case apply(F, A) of
        {ok, Tokens, Endline} ->
      yeccpars1(Tokens, {{F, A}, Endline}, State, States, Vstack);
        {eof, Endline} ->
            yeccpars1([], {no_func, Endline}, State, States, Vstack);
        {error, Descriptor, _Endline} ->
            {error, Descriptor}
    end;
yeccpars1([], {no_func, no_line}, State, States, Vstack) ->
    Line = 999999,
    yeccpars2(State, '$end', States, Vstack, yecc_end(Line), [],
              {no_func, Line});
yeccpars1([], {no_func, Endline}, State, States, Vstack) ->
    yeccpars2(State, '$end', States, Vstack, yecc_end(Endline), [],
              {no_func, Endline}).

%% yeccpars1/7 is called from generated code.
%%
%% When using the {includefile, Includefile} option, make sure that
%% yeccpars1/7 can be found by parsing the file without following
%% include directives. yecc will otherwise assume that an old
%% yeccpre.hrl is included (one which defines yeccpars1/5).
yeccpars1(State1, State, States, Vstack, Token0, [Token | Tokens], Tzr) ->
    yeccpars2(State, element(1, Token), [State1 | States],
              [Token0 | Vstack], Token, Tokens, Tzr);
yeccpars1(State1, State, States, Vstack, Token0, [], {{_F,_A}, _Line}=Tzr) ->
    yeccpars1([], Tzr, State, [State1 | States], [Token0 | Vstack]);
yeccpars1(State1, State, States, Vstack, Token0, [], {no_func, no_line}) ->
    Line = yecctoken_end_location(Token0),
    yeccpars2(State, '$end', [State1 | States], [Token0 | Vstack],
              yecc_end(Line), [], {no_func, Line});
yeccpars1(State1, State, States, Vstack, Token0, [], {no_func, Line}) ->
    yeccpars2(State, '$end', [State1 | States], [Token0 | Vstack],
              yecc_end(Line), [], {no_func, Line}).

% For internal use only.
yecc_end({Line,_Column}) ->
    {'$end', Line};
```

HUH?

Functions are Data

PEG Functions are Higher-Order

Higher-Order Functions in Parsing

- `p_optional(fun e/2) -> fun(Input, Index)`
- Receives current input and index/offset
- Returns `{fail, Reason}` or `{Result, Rest, NewIndex}`

Combinators vs. Parsers

```
p_optional(fun e/2) -> fun(Input, Index)
```

How to Design a Parser

Recursive Descent

- Functions call other functions to recognize and consume input
- Backtrack on failure and try next option

Predictive Recursive Descent

- Functions call other functions to recognize and consume input
- Stream lookahead to determine which branch to take (firsts, follows)
- Fail early, retry very little

Recursive Descent

O(2^N)

Predictive Parsers are Expensive to Build

"Packrat" Parsers

O(N)

"Packrat" Parsers - O(N)

- Works and looks like a Recursive Descent Parser
- Memoizes (remembers) intermediate results - success and fail
- Trades speed for memory consumption

Memoization is a HO Parsing Function

```
memoize(Rule, Input, Index, Parser)->
{fail, Reason} | {Result, Rest, NewIndex}
```

```
memoize(Rule, Input, Index, Parser) ->
  case get_memo(Rule, Index) of
    undefined ->
      store_memo(Rule, Index, Parser(Input, Index));
    Memoized -> Memoized
  end.
```

How Memoization Helps

```
if_stmt <- “if” expr “then” stmt “else” stmt /
           “if” expr “then” stmt
```

if_stmt <- “if” expr “then” stmt “else” stmt /
“if” expr “then” stmt

if A then if B then C else D

if_stmt <- “if” expr “then” stmt “else” stmt /
“if” expr “then” stmt

if A then if B then C else D

if_stmt <- “if” expr “then” stmt “else” stmt /
“if” expr “then” stmt

if A then if B then C else D

if_stmt <- “if” expr “then” stmt “else” stmt /
“if” expr “then” stmt

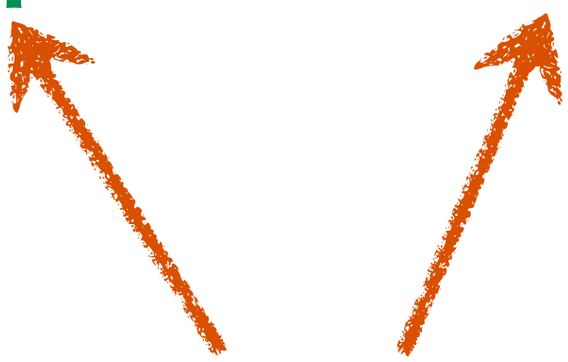
if A then if B then C else D

if_stmt <- “if” expr “then” stmt “else” stmt /
“if” expr “then” stmt

if A then if B then C else D

if_stmt <- “if” expr “then” stmt “else” stmt /

“if” expr “then” stmt



memoized!

if A then if B then C else D

Enter Neotoma

http://github.com/seancribbs/neotoma

Neotoma

- Defines a metagrammar
- Generates packrat parsers
- Transformation code inline
- Memoization uses ETS
- Parses and generates itself

Let's build a CSV parser

```
rows <- row (crlf row)* / '';
```

```
rows <- row (crlf row)* / '';
row <- field (field_sep field)* / '';
```

```
rows <- row (crlf row)* / '';
row <- field (field_sep field)* / '';
field <- quoted_field / (!field_sep !crlf .)*;
```

```
rows <- row (crlf row)* / '';
row <- field (field_sep field)* / '';
field <- quoted_field / (!field_sep !crlf .)*;
quoted_field <- '"' ('""' / !'"' .)* '"';
```

```
rows <- row (crlf row)* / '';
row <- field (field_sep field)* / '';
field <- quoted_field / (!field_sep !crlf .)*;
quoted_field <- '"' ('""' / !'"' .)* '"';
field_sep <- ',';
crlf <- '\r\n' / '\n';
```

```
rows <- head:row tail:(crlf row)* / '';
row <- head:field tail:(field_sep field)* / '';
field <- quoted_field / (!field_sep !crlf .)*;
quoted_field <- '"' string:('""' / !'"' .)* '"';
field_sep <- ',';
crlf <- '\r\n' / '\n';
```

```
rows <- head:row tail:(crlf row)* / ‘’
`
case Node of
   [] -> [];
   [“”] -> [];
      ->
      Head = proplists:get_value(head, Node),
      Tail = [Row || [_CRLF,Row] <-
                     proplists:get_value(tail, Node)],
      [Head|Tail]
end
`;
```

```
1> neotoma:file(“csv.peg”).
ok
2> c(csv).
{ok, csv}
```

DO IT LIVE!

FP + PEG + Packrat = WIN

- Clear relationship between grammar and generated code
- Powerful abstractions, declarative code
- Simpler to understand and implement
- Terminals *in* context, no ambiguity

```
sean <- question*;
```

sean <- question*;

sean@basho.com
@seancribbs