Divergence Parser Combinators

Spencer Tipping

August 18, 2010

Contents

1 Introduction

The Divergence parser combinator library is a port of the Scala or Haskell parser combinators into JavaScript. The notation benefits from using Rebase (thus enabling operator overloading), but it isn't necessary. Each parser method has both an identifier name and optionally an operator name.

Listing 1 divergence.parser.js

```
d.rebase (function () {
  var p = d.parser = '@matcher = $0'.ctor ({
      fn: '@bound("match")'.fn(),
    fail: '{failure: $0}'.fn(),
    pass: '{stream: $0, result: $1}'.fn(),
   match: '@matcher.apply(this, @_)'.fn()});

  d.rebase.alias_in (d.parser.prototype, {
    '%$%': 'fail',
      '%': 'match'});

  d.functional(p.prototype);
```

2 Streams

A stream is an immutable list of objects. In this case, a stream is an indexed proxy over a string. It must support the `head()` and `tail()` methods, where `tail()` should return a new stream minus the head. Technically this could all be done using strings directly, but the head/tail notation is convenient.

Listing 2 `divergence.parser.js (continued)`

```
p.indexed_stream = (index, empty) >$> '@xs = $0, @position = $1 || 0'.ctor ({
  empty: '$0.call(@xs, @position)'.fn(empty.fn()),
   head: '$0.call(@xs, @position)'.fn(index.fn()),
   tail: 'new @constructor (@xs, @position + 1)'.fn()});

p.empty_stream = {empty: _ >$> true};

p.string_stream = p.indexed_stream ('@charAt($0)', '$0 >= @length');
p.array_stream  = p.indexed_stream ('$_[$0]',      '$0 >= @length');
p.list_stream   =
  '@h = $1, $t = $2.empty ? $2 : new $0.list_stream ($2, $0.empty_stream)'.fn (p).ctor ({
  empty: _ >$> false, head: '@h'.fn(), tail: '@t'.fn()});
```

3 Parsers

Parsing a CFG is a recursive process. There are base cases (i.e. terminal elements) and operators on parsers. The basic idea is that a parser will return one of two things. It can return an object of the form `{stream, result}` to indicate success (in which case the stream is the original minus whatever input was consumed), or an object of the form `{failure, ...}` to indicate a failure state.

3.1 Parser extensions

These functions provide a uniform way to extend the parser class with custom combinators.

Listing 3 `divergence.parser.js (continued)`

```
var def_combinator = (name, operator, matcher) >$>
  (p[name] = matcher,
   p.prototype[name] = p.prototype[operator] =
     'new @constructor($0($_, $1))'.fn (matcher));
```

3.2 Terminal parser

This is a simple parser to match a terminal character. For coding simplicity, I'm just providing one for strings, though you could easily write one for other data types given a suitable equality operator.

Listing 4 divergence.parser.js (continued)

```
p.terminal = match >$> (stream >$>
  (! stream.empty() && stream.head() === match ?
    this.pass (stream.tail(), match) :
    this %$% 'Expected "#{match}" but found (#{stream.head()})'));
```

It is also useful to have a parser that matches on end of input:

Listing 5 divergence.parser.js (continued)

```
p.end = stream >$>
  (stream.empty() ? this.pass (stream, null) :
    this %$% 'Expected end of input, but found #{stream.head()}');
```

3.3 Sequence combinator

This parses one element and then another. It fails if either parser fails. If both parsers succeed, the result is a `list_stream` of the result from each one. If the second is not a stream, then it is consed into a stream that is consed onto the empty stream for uniformity.

Listing 6 divergence.parser.js (continued)

```
def_combinator ('sequence', '<<', (p1, p2) >$> (stream >$>
  ((p1 % stream, this) |$> ((m1, t) >$>
    (m1.failure ? t %$% m1.failure :
      (p2 % m1.stream |$> (m2 >$>
        (m2.failure ? t %$% m2.failure :
          this.pass (m2.stream, new p.list_stream (m1.result, m2.result))))))))));
```

3.4 Disjunctive combinator

This parses one element using two alternatives. If the first fails, then the second is used. If the second fails, then neither is used and the disjunction fails. The result is as returned from whichever parser was successful.

Listing 7 divergence.parser.js (continued)

```
def_combinator ('disjunction', '|', (p1, p2) >$> (stream >$>
  ((p1 % stream, this) |$> ((m1, t) >$>
    (m1.stream ? m1 :
      (p2 % stream |$> (m2 >$>
        (m2.stream ? m2 :
          this %$% m2.failure)))))))));
```

3.5 Mapping combinator

Provides an opportunity to change the result value by mapping it through a function. Because of the parser-function isomorphism, you can use this on another parser to achieve composition.

Listing 8 divergence.parser.js (continued)

```
def_combinator ('map', '*', (p, f) >$> (stream >$>
  ((p % stream, this) |$> ((m, t) >$>
    (m.failure ? t %$% m.failure :
      this.pass (m.stream, f.fn()(m.result)))))));
```

Listing 9 divergence.parser.js (continued)

```
}) ();
```