MPC Manual

Max Rottenkolber

Sunday, 26 July 2015

Table of Contents

1 Abstract

MPC is a monadic parser combinators library. It provides a toolbox of predefined parsers, facilities for error handling and parses arrays, streams and lists.

This manual summarizes the exported functions of the packages `mpc`, `mpc.characters`, and `mpc.numerals`. Refer to the *MPC API* (`api.html`) for detailed descriptions of all exported symbols. To learn more about MPC's internals and monadic parser combinators in general read *Drew Crampsie's parser combinators tutorial* (`https://github.com/drewc/smug/blob/master/smug.org`).

2 A brief practical example

Assuming you want to parse an email address in the form of `<user>@<host>`. Let's start by defining our package:

```
(defpackage simple-address
  (:use :cl :mpc :mpc.characters))

(in-package :simple-address)
```

Next we restrict the allowed characters in the *user* and *host* fields:

```
(defun =address-character ()
  (=or (=satisfies #'alphanumericp)
       (=one-of '(#\- #\_ #\. #\+))))
```

That is: Alphanumeric characters or any one of the *dash*, *underscore*, *perioid* and *plus* chracters. Note how we use Common Lisp's `alphanumericp`.

Finally we use `=address-character` to implement a simple address parser:

```
(defun =simple-address ()
  (=let* ((user (=string-of (=address-character)))
          (_ (=character #\@))
          (host (=string-of (=address-character))))
    (=result (list user host))))
```

The _ binding in the `=let*` bindings is used to ignore the `@` seperator. We return a list containing the `user` and `host` strings using `=result`.

We can now apply our grammar using `run`:

```
(run (=simple-address) "foo@example.com")
 ⇒ ("foo" "example.com")

(run (=simple-address) "!!!@@@.com")
 ⇒ NIL
```

3 `run`: Main entry point.

`run` is the main entry point to MPC and has to be used to run a *parser* against an *input-source*. The *input-source* can be of type `array`, `file-stream` or `list`. Because MPC supports non-deterministic parsers which can return multiple results, `run` accepts a keyword parameter *result*, a function used to select the desired return value. By default `run` returns only the first result's value.

4 Primitive parsers and combinators

The core of MPC is made up of primitive parsers and combinators. A parser is a function that accepts an input source and returns a list of pairs containing the result and the remaining input if it is successful and `nil` othwerise to signal its failure to parent parsers. A combinator is a function which returns a parser. For consistency primitive parsers are defined as combinators that that always return the primitive parser.

`=item` is used to pop off an item from the input. It fails if the input is empty. `=result` always succeeds with a given value without consuming input. It is used to return arbitrary values from a parser. To check for end of input there is `=end-of-input` which succeeds only when the input is empty.

The primitive combinator `=bind` permits applying parsers in sequence and offers a way to access their intermediate results. `=plus` lets us combine parsers in a non-deterministic way while `=or` and `=and` are deterministic alternatives. `=if` allows for conditional application of parsers.

5 `=let*`: Syntax for lispers

The `=let*` macro offers a lispy syntax for `=bind`. It binds the results of a sequence of parsers to variables and unless any parser fails runs the body parsers. It also understands the special symbol _ (underscore) to signify *ignorable* bindings, where ignorable means that no symbol shall be bound to the value of a given parser (`=let*` nevertheless requires the parser to succeed).

The syntax of `=let*` is as follows:

`(=let* ((`*symbol parser*`)*) `*parser*`*)`

6 Error handling

`=fail` simply always fails. It optionally accepts expressions to be evaluated at failure. Those expressions are permitted to call `get-input-position`, which can be used to determine where a failure occurred.

Two other error handling facilities `=handler-case` and `=restart-case` do what their names suggest. Instead of forms to be evaluated, every *case clause* accepts parsers to be run.

7 More combinators

`=when` and `=unless` behave like `=if` with an implicit `progn` and no *else clause*. Just like `when` and `unless` in *Common Lisp.*

`=not` takes one parser and, if it would *fail*, consumes and returns the next item from input.

`=prog1` and `=prog2` behave like `=and` but return the result of the first or second parser respectively. `=maybe` applies a parser and succeeds even if the parser fails. `=list` also behaves like `=and` but collects all results in a list.

`=satisfies` applies `(=item)` but only succeeds when the result returned by `(=item)` satisfies a given predicate. `=eql`, `=one-of`, `=none-of` and `=range` are like `=satisfies` but require the result returned by `(=item)` to be `eql` to a given value, `eql` to one or none of the values in a given list or to be inside a range defined by a predicate and a lower and upper bound respectively.

`=one-or-more`, `=zero-or-more`, `=one-to`, `=zero-to`, `=at-least` and `=exactly` do as their names suggest and apply parsers multiple times in various variants.

`=funcall` applies a parser and returns the result of a given function called on the parsers result.

8 Parsers for character input

`=character` behaves like `=eql` but uses `char=` instead of `eql`. `=string` is similar too as it parses a given string or fails. `=string-of` parses a string of characters parsed by a given parser.

`=whitespace` and `=skip-whitespace` parse or skip characters commonly considered as whitespace respectively. `=newline` parses the newline character and `=line` parses a string of characters terminated by newline or end of input.

9 Parsers for numerals

`=digit` parses a number from a digit character. `=natural-number` and `=integer-number` both parse numbers from numeral strings while the latter also understands a leading dash for negativity. All three parses accept an optional radix argument.