compynator

google

Nov 05, 2020

CONTENTS

This is index.

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER ONE

REFERENCE

1.1 compynator.core

The essentials of all parser combinators.

The six basic regex definitions are mapped according to:

Regex	Compynator
empty set	`Fail`
epsilon	`Empty`
character	`Terminal`
concatenation	+
alternative	| or ^
Kleene star	`repeat`

Monadic properties are `Succeed` for unit, `Parser.then` for bind, and optionally `Fail` for zero.

`compynator.core.`**`Empty = Parser<Succeed@140281776558224>`**

An empty string. This always succeeds.

`class` `compynator.core.`**`Failure`**(*parser*, *message*, *remain=''*, *cause_or_causes=None*)

A collection of zero ``Result``s.

This class is used to propagate parse failures and incrementally construct a `Success`. We usually start out with an instance of this class, then add more `Result` objects to it to produce a `Success`.

```
>>> parser = 'a' + Terminal('b') + 'c'
>>> ret = Failure(parser, 'Parser fails.', None)
>>> s = ret.add_all(parser('abc'))
>>> isinstance(s, Success)
True
```

`add`(*result*)

Returns a `ResultSet` (could be `self`) with `result`.

`add_all`(*results*)

Returns a `ResultSet` (could be `self`) with all `results`.

class compynator.core.**ParseContext**(*options=None*)

Internal book-keeping data structure.

class compynator.core.**ParseOptions**(*max_recursion*)

property max_recursion

Alias for field number 0

class compynator.core.**Parser**(*parser_function*)

Callable that takes a sequence of tokens & returns a ResultSet.

The specific types of inputs and outputs are not known. However, inputs usually are strings. The requirements for inputs are:

1. they must have __len__
2. they are indexable and slicable

A parser must return a collection of Result``s. Their ``value elements can be any type but their remain elements must be a slice of the input tokens.

This class can be used as a decorator:

```
@Parser
def head(tokens):
    if len(tokens) >= 1:
        return Success(result=Result(tokens[0], tokens[1:]))
    return Failure(head, 'Unable to obtain more tokens', tokens)
```

In that example, head is a parser that returns the first element of the sequence of tokens. Then head can be chained (then, +) with other parsers, filtered (where, value), or composed together to be more useful.

call(*callback*)

Simple wrapper around filter to always call callback.

In ambiguous grammar (like the example below), there might be repeated results if call makes up a part of the variable. Please note the difference in two definitions of the same production rule.

```
>>> count = 0
>>> def cb(r):
...     global count
...     count += 1
>>> empty = Succeed('')
>>> s = ((Terminal('s') + (lambda _: s)) ^ empty).call(cb)
>>> r = s('ss')
>>> assert len(r) == 3
>>> count
6
>>> count = 0
>>> s = (Terminal('s') + (lambda _: s)) ^ empty
>>> r = s.call(cb)('ss')
>>> assert len(r) == 3
>>> count
3
```

filter(*callback*, *take_if=True*)

Executes callback on a successful parse and filters results.

callback must take a Result. Every possible result of a rule will be passed to callback.

If truth value as returned by callback is the same as take_if, that Result object is included.

NOTE: The ordering between `filter` and `memoize` is important and may result in `callback` not being invoked.

memoize()

Memoizes parsed results of `self`.

The memoization allows for ambiguous grammar to be processed efficiently. See the paper Parser Combinators for Ambiguous Left Recursive Grammars.

This modifier is recommended when the unbiased `__xor__` operator is used, or when left recursion is in the grammar:

```
>>> empty = Succeed('')
>>> s = ((Terminal('s') + (lambda _: s) + (lambda _: s)) ^
...      empty).memoize()
>>> len(s('s' * 20))
21
```

Without the `memoize` modifier in the above example, it would take a very long time to parse.

parse(*tokens*)

Parses the input `tokens` under the default context.

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

repeat(*lower=0*, *upper=None*, *reducer=<built-in function concat>*, *value=''*, *take_all=False*)

Repeatedly parses [lower, upper] occurrences.

If `upper` is `None`, there is no upper bound. The `reducer` is used to join the results together similar to how it is used in `then`. The zeroth parse result (`parser` is not invoked yet) is a `Success` of `value`. The first reduction is between zeroth and first results. If `take_all`, then all results are returned. If not `take_all`, then only the greediest results are returned.

```
>>> p = Terminal('a').repeat()
>>> set(p(''))
{Result(value='', remain='')}
>>> set(p('b'))
{Result(value='', remain='b')}
>>> set(p('a'))
{Result(value='a', remain='')}
>>> set(p('aa'))
{Result(value='aa', remain='')}
```

skip(*binder*)

Similar to `then`, but the reducer takes the first value.

then(*binder*, *reducer=<function Parser.<lambda>>*)

Chains `self` and parser(s) returned by `binder` via `reducer`.

This is the `bind` function in monadic sense. `binder` is a callable that takes in a `Result.value` and returns a `Parser` object. This parser is then applied on `Result.remain`.

`binder` can also be a `Parser` object. In this case, `binder` is used directly as the second parser.

If not, `binder` will be converted into a `Terminal(str(binder))`.

`reducer` takes two arguments, the first is `Result.value` of this parser, and the second is the `Result.value` of the second parser. The result of `reducer` makes up the final result of the composed parser.

The default `reducer` only takes the second `Result.value`.

In code, this looks like:

```
ret = Fail(tokens)
for value, remain self(tokens):
    next_parser = binder(value)
    for next_value, next_remain in next_parser(remain):
        final_value = reducer(value, next_value)
        ret = ret.add(Result(final_value, next_remain))
```

value (*converter_or_value*)

Converts `Result.value` into a different value.

`converter_or_value` can be a callable, or an object. If it is a callable, it takes `Result.value` and returns a converted value. If it is a value, that value is used.

For example:

```
>>> digit = One.where(lambda c: '0' <= c <= '9')
>>> set(digit('8bc'))
{Result(value='8', remain='bc')}
>>> digit_as_int = digit.value(int)
>>> set(digit_as_int('8bc'))
{Result(value=8, remain='bc')}
```

where (*predicate*)

Selects results whose values pass `predicate`.

`predicate` is a callable that takes `Result.value` and returns `True` if that `Result` should be included. This is a convenient wrapper around `filter`.

For example:

```
>>> digit = One.where(lambda c: '0' <= c <= '9')
>>> set(digit('abc'))
set()
>>> set(digit('8bc'))
{Result(value='8', remain='bc')}
```

class `compynator.core.`**Result**

Holds the parsed results.

Each result is a 2-tuple of value and remaining unparsed sequence of tokens.

NOTE: The input tokens are assumed to be immutable and `len(remain)` is sufficient to tell if two ``Result.remain``s are equal.

class `compynator.core.`**ResultSet**

A sized iterable collection of `Result`.

To incrementally construct a result set, first start with a `Failure`, then add more `Result` via `add` or `add_all`.

add (*result*)

Returns a `ResultSet` (could be `self`) with `result`.

add_all (*results*)

Returns a `ResultSet` (could be `self`) with all `results`.

class `compynator.core.`**Succeed** (*value*)

Always returns a parsed result of `value` regardless of input.

For example:

```
>>> s = Succeed(10)
>>> set(s('abc'))
{Result(value=10, remain='abc')}
>>> set(s('def'))
{Result(value=10, remain='def')}
```

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

class compynator.core.**Success**(**args*, *result=None*, *results=None*)

A collection of `Result` in a successful parse.

A `Success` must have at least one `Result`. The constructor can take either keyword argument `result` or `results`, but not both at the same time.

add(*result*)

Returns a `ResultSet` (could be `self`) with `result`.

add_all(*results*)

Returns a `ResultSet` (could be `self`) with all `results`.

class compynator.core.**Terminal**(*terminal*)

Matches `terminal` to the beginning of input tokens.

```
>>> t = Terminal('t')
>>> set(t(''))
set()
>>> set(t('t'))
{Result(value='t', remain='')}
```

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

compynator.core.**default_parse_context**(*tokens*)

Returns `ParseContext` for `tokens`.

1.2 compynator.niceties

compynator.niceties.**Alnum = Parser<_Or@140281776574672>**

Exactly one ASCII letter or digit.

compynator.niceties.**Alpha = Parser<_Or@140281776574608>**

Exactly one letter a-zA-Z

class compynator.niceties.**Collect**(**parsers*)

A combinator that runs through all `parsers` in sequence and collects their results in a collection of many flattened collections.

This is best described with examples:

```
>>> a, b, c = [Terminal(x) for x in 'abc']
>>> p = Collect(a, b, c)
>>> set(p('adc'))
set()
>>> p('adc')
Failure('Failed to collect.', 'adc', [Failure("Parser index 1: Expecting terminal
↪'b'.", 'dc', ())])
```

(continues on next page)

(continued from previous page)

```
>>> set(p('abc'))
{Result(value=('a', 'b', 'c'), remain='')}
>>> a = a.repeat(0, 2, take_all=True)
>>> p = Collect(a, a, a)
>>> rs = p('a')
>>> len(rs)
4
>>> Result(value=('', '', ''), remain='a') in rs
True
>>> Result(value=('', '', 'a'), remain='') in rs
True
>>> Result(value=('', 'a', ''), remain='') in rs
True
>>> Result(value=('a', '', ''), remain='') in rs
True
>>> len(p('aa'))  # -/-/-, -/-/a, -/a/-, a/-/-,         ...               # -/-/
↪aa, -/a/a, -/aa/-, a/-/a, a/a/-, aa/-/-
10
```

Note that the final `ResultSet` could grow exponentially.

`parse_with_context`(*tokens*, *context*)
Parses input `tokens` under the context of `context`.

`compynator.niceties.`**`Digit = Parser<_Filter@140281776561360>`**
Exactly one decimal digit.

`class` `compynator.niceties.`**`Forward`**
A forward declaration of a rule.

This is useful in case the rule is defined recursively. For example, the BNF rule `exp ::= (exp '-' exp) | 'o'` could be defined as followed:

```
>>> exp = Forward()
>>> exp.is_(((exp + '-' + exp) ^ 'o').memoize())
>>> set(exp('o'))
{Result(value='o', remain='')}
>>> sorted(exp('o-o'))
[Result(value='o', remain='-o'), Result(value='o-o', remain='')]
```

A forward declaration of Parser is the same as referring to that parser in a lambda:

```
>>> exp = (Succeed(None).then(lambda _: exp + '-' + exp) ^ 'o').memoize()
>>> set(exp('o'))
{Result(value='o', remain='')}
>>> sorted(exp('o-o'))
[Result(value='o', remain='-o'), Result(value='o-o', remain='')]
```

A `RuntimeError` will be raised if a `Forward` has not called `is_`, or if that method is called more than once.

```
>>> exp = Forward()
>>> exp('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: A forward declaration has no definition.
>>> exp.is_('abc')
>>> exp.is_('abc')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in ?
RuntimeError: Already defined.
```

is_(*parser*)

Defines a forward declaration.

If `parser` is not typed `Parser`, its string representation will be made into a `Terminal`.

This method must be called exactly once for each `Forward` object. A `RuntimeError` will be raised if it is called more than once.

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

`compynator.niceties.`**HexDigit = Parser<_Or@140281776561616>**

Exactly one hexadecimal digit.

class `compynator.niceties.`**ITerminal**(*terminal*)

Case insensitive terminal.

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

class `compynator.niceties.`**Lookahead**(*parser*, *take_if=True*, *value=''*)

Tries `parser` but does not consume input.

If the truth value of the parse result is `take_if`, a `Success` of `value` is returned. Otherwise, a `Failure` is returned.

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

`compynator.niceties.`**Lower = Parser<_Filter@140281776562000>**

Exactly one letter a-z.

`compynator.niceties.`**OctDigit = Parser<_Filter@140281776561808>**

Exactly one octadecimal digit.

class `compynator.niceties.`**Regex**(*regex*)

Regex matcher.

parse_with_context(*tokens*, *context*)

Parses input `tokens` under the context of `context`.

`compynator.niceties.`**Upper = Parser<_Filter@140281776574544>**

Exactly one letter A-Z

PYTHON MODULE INDEX

INDEX