kessel Documentation

Release 0.2.0

Tony Young

August 20, 2015

Contents

1 Contents **3**
- 1.1 Tutorial . . . 3
- 1.2 API Documentation . . . 6

2 Indices and tables **11**

Python Module Index **13**

Kessel is a parser library for Python that supports parsing a wide variety of grammars, from simple LL(1) grammars to complex, context-sensitive grammars with minimal hassle.

CHAPTER 1

Contents

1.1 Tutorial

The following sections are tutorials on getting started with Kessel. Hopefully, they should cover the whole feature set of Kessel.

1.1.1 Hello, Kessel!

Getting started with Kessel is easy! Suppose we wanted to parse greetings in the form `"hello, [name]!"` We can use Kessel's parser facilities to operate on the string directly:

```
>>> from kessel import *
>>> greeting = literal("hello, ") >> many1(none_of("!")) << literal("!")
```

The operators >> and << are overloaded – they point in the direction of the output you want to keep, i.e. in this case, the parse result of `"hello, "` and `"!"` would be discarded.

We can use this new `greeting` parser:

```
>>> greeting.parse("hello, tony!")
['t', 'o', 'n', 'y']
```

That's kind of annoying. We can use `mapf` to specify a function to run after parsing to fix that:

```
>>> name = mapf(many1(none_of("!")))("".join)
>>> greeting = literal("hello, ") >> name << literal("!")
>>> greeting.parse("hello, tony!")
'tony'
```

What if we wanted to customize the greeting?

```
>>> def greeting(salutation="hello"):
...     return literal(salutation + ", ") >> name << literal("!")
```

Now, if we're Australian:

```
>>> greeting("g'day").parse("g'day, tony!")
'tony'
```

But we still don't understand our Stateside friends:

```
>>> greeting("g'day").parse("howdy, tony!")
Traceback (most recent call last):
...
kessel.parser.Unexpected: expected one of 'g', got 'h' at index 0
```

So we can use the `choice` combinator:

```
>>> def greeting(salutations=("hello",)):
...     start = choice(*[literal(salutation) for salutation in salutations])
...     return start >> literal(", ") >> name << literal("!")
```

And now:

```
>>> greeting(("g'day", "howdy")).parse("howdy, tony!")
'tony'
>>> greeting(("g'day", "howdy")).parse("g'day, tony!")
'tony'
```

1.1.2 Reverse Polish Notation Calculator

Hopefully, the first tutorial was relatively straightforward. Now, we can move onto a slightly more complex example — a Reverse Polish notation calculator. For this, we can use `gen_parser`, which provides a way for us to handle parser input in Python generators.

If you're not familiar with Reverse Polish notation, it works like this:

- If a number is encountered, it is pushed onto the stack.
- If an operator is encountered, it pops the top two numbers off the stack and runs the operator on them.

For example:

```
> 2 2 +
4

> 2 2 3 + *
10
```

To parse this, we can define a few things upfront:

```
>>> from kessel import *
>>> from operator import add, sub, mul, floordiv
>>> wspace = optional(word(" \n\r\t"))
>>> operator = mapf(one_of(*"+-*/"))({
...     "+": add,
...     "-": sub,
...     "*": mul,
...     "/": floordiv
... }.__getitem__)
>>> number = mapf(word("0123456789"))(int)
```

Now, we can write our parser.

```
@gen_parser
def rpn():
    """
    This is a really simple Reverse Polish Notation parser that also does
    evaluation.
    """
```

```
    stack = []

    while True:
        # Parse any preceding whitespace.
        yield wspace

        # Try parse an operator first.
        try:
            op = yield operator
        except Unexpected:
            pass
        else:
            stack.append(op(stack.pop(), stack.pop()))
            continue

        # If that fails, try parse a number.
        try:
            n = yield number
        except Unexpected:
            pass
        else:
            stack.append(n)
            continue

        # Otherwise, this has to be the end of input.
        yield eof
        break

    return stack
```

We can use `yield` to yield to a parser, then the value it returns will be yielded back into the calling parser. In this example, we can easily get operators and numbers from the input stream without having to mess with it directly. We can run our parser now to see if it works:

```
>>> rpn.parse("2 2 +")
[4]
>>> rpn.parse("2 2 3 + *")
[10]
```

1.1.3 Lexing

While Kessel can operate directly on strings, it gives questionable parser error messages. While it's useful for something quick and dirty, for something more robust coupling a lexer to Kessel may be more useful.

1.1.4 Gotchas

- Kessel can only parse iterators with items that support `hash` and ==. This means Kessel works out-of-the-box to parse iterators of strings, and you can easily implement `hash` and == for lexer output so it can be fed into Kessel parsers.
- Lookahead by default is LL(1). If infinite lookahead is required, you need to wrap the parser in a `try_` parser. There are no options for other lookaheads.
- Try to avoid using `try_`, as it will tee the iterator. This means that Kessel may exhibit a lot of memory usage, possibly parsing to the end of the entire iterator, before attempting another alternative.

- If you want the parser to consume the entire string, you need to expect an `eof` for end of input. Otherwise, Kessel will succeed if the start of the string matches and may not consider the whole string.
- In `gen_parser` parsers, performance may be better if complex parsers are constructed outside of the generator function, as these parsers will need to be constructed every time if placed inside the generator. In the case of context-sensitive parsers, you probably don't have much of a choice.

1.2 API Documentation

1.2.1 `kessel.primitives` — Base parsers

Primitive parsers.

The functions contained are primitive functions for constructing parsers; for creating *parsing functions*.

A parsing function is any function that:

- takes an iterator,
- on success, returns the value parsed and an iterator, and,
- on failure, raises an instance of *ParseError*.

You must _always_ use the iterator returned from parsing functions to continue parsing (or to otherwise consume input).

exception `kessel.primitives.`**`ParseError`**(*value*, *it*, *expected*)
Describes an error that occurs during parsing.

Parameters

- **`value`** – The erroneous item in the stream encountered.
- **`it`** – The remainder of the stream.
- **`expected`** – A set of expected values. This is intended only as a human-readable description.

class **`Entity`**(*value*)
Defines an entity used to display in the error message.

`__weakref__`
list of weak references to the object (if defined)

`ParseError.`**`__weakref__`**
list of weak references to the object (if defined)

`ParseError.`**`unpeek`**()
Returns the iterator with the erroneous item, prepended.

This is useful when the consumption of a single item in the stream needs to be undone. This will also drop sentinel EOF values from the stream.

class `kessel.primitives.`**`Proxy`**
Proxies a parsing function such that it can be defined later.

This is useful in the case of recursive grammars, where a production rule may be defined in terms of itself.

`__call__`(*it*)
Forwards the stream to the underlying parser.

Parameters **`it`** – The stream to parse.

Raises ValueError If there is no parsing function bound.

Returns The parse result.

__init__()
Constructor.

__weakref__
list of weak references to the object (if defined)

bind(*parser*)
Binds a parsing function to the proxy.

This can only be done once.

Parameters **parser** – A parsing function to bind.

Raises ValueError If there is already a parsing function bound.

kessel.primitives.**error**(*expected*)
Matches nothing, and will throw an error with the specified expectation.

Parameters **expected** – List of expected values appropriate here.

Returns The parsing function.

kessel.primitives.**match**(*cond*, *expected=None*)
Advances the input stream once and matches against a condition.

If the condition does not hold, then *ParseError* is thrown.

If end-of-file is reached, *ParseError* is also thrown, but the value contained in the error will be the special EOF literal.

Parameters

- **cond** – Boolean predicate that the item in the stream should match.
- **expected** – List of expected values appropriate here.

Raises ParseError If the predicate did not hold.

Returns The parsing function.

kessel.primitives.**unit**(*f*)
Returns the result of the given function without advancing the stream.

Parameters **f** – The function to be run.

Returns The parsing function.

1.2.2 kessel.combinators — Parser combinators

Parser combinators.

These functions combine parsers together to create other parsers.

kessel.combinators.**between**(*first*, *last*, *parser*)
Runs a parser between two other parsers.

Parameters

- **first** – First parser to run.
- **last** – Last parser run.
- **parser** – The parser to run whose result will be returned.

Returns The parsing function.

kessel.combinators.**choice**(**parsers*)
Tries each parser specified in order.

Please note this parser is predictive – it will only try alternatives if the previous parser did not consume any input.

Parameters **parsers** – A series of parsers to attempt.

Returns The parsing function.

kessel.combinators.**count**(*n*, *parser*)
Runs a parser a specified number of times.

Parameters

- **n** – Number of times to run the parser.
- **parser** – The parser to run.

Returns The parsing function.

kessel.combinators.**many**(*parser*)
Runs a parser zero or more times.

Parameters **parser** – The parser to run.

Returns The parsing function.

kessel.combinators.**many1**(*parser*)
Runs a parser one or more times.

Parameters **parser** – The parser to run.

Returns The parsing function.

kessel.combinators.**mapf**(*parser*)
Creates a decorator for a parser to specify custom actions.

Parameters **parser** – Parser to decorate with.

Returns A decorator for a function that must take the parser result and returns the intended output result.

kessel.combinators.**mapf_star**(*parser*)
Creates a decorator for a parser to specify custom actions (unpacked).

Parameters **parser** – Parser to decorate with.

Returns A decorator for a function that must take the unpacked parser result (which must be a list) and returns the intended output result.

kessel.combinators.**not_followed_by**(*parser*, *expected=None*)
Performs a negative lookahead.

Parameters **parser** – The parser that should not succeed.

Returns The parsing function.

kessel.combinators.**option**(*parser*, *default=<function <lambda>>*)
Runs a parser optionally.

If the parser could not be applied, the default value function will be called and its result will be returned. It will only fail if input was consumed.

Parameters

- **parser** – The parser to run.
- **default** – Function providing the default value.

Returns The parsing function.

kessel.combinators.**sep_by**(*sep*, *parser*)
Runs a parser zero or more times, with items separated by another parser.

Parameters

- **sep** – Separator parser.
- **parser** – The parser to run.

Returns The parsing function.

kessel.combinators.**sep_by1**(*sep*, *parser*)
Runs a parser one or more times, with items separated by another parser.

Parameters

- **sep** – Separator parser.
- **parser** – The parser to run.

Returns The parsing function.

kessel.combinators.**sequence**(**parsers*)
Runs parsers in sequence.

Parameters **parsers** – A series of parsers to run in sequence.

Returns The parsing function.

kessel.combinators.**sequence_l**(*l*, *r*)
Runs two parsers in sequence, preserving only the result of the former.

Parameters

- **l** – Parser whose result should be preserved.
- **r** – Parser whose result should be discarded.

Returns The parsing function.

kessel.combinators.**sequence_r**(*l*, *r*)
Runs two parsers in sequence, preserving only the result of the latter.

Parameters

- **l** – Parser whose result should be discarded.
- **r** – Parser whose result should be preserved.

Returns The parsing function.

kessel.combinators.**suppress_expect**(*parser*)
Suppress any expectations from parsing errors.

Parameters **parser** – The parser to run.

Raises ParseError If the parser fails. It will have all expectations stripped.

Returns The parsing function.

kessel.combinators.**try_**(*parser*)

Wraps a parser such that on error, it is reset to the original position in the stream.

Please note that usage of this function will tee the stream, which may require significant auxiliary storage, so it is best avoided if necessary.

Parameters **parser** – Parser to wrap.

Returns The parsing function..

Raises **ParseError** If the underlying parser encounters an error. However, the error is re-wrapped such that the stream is reset to where the parser was first attempted.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

k

kessel.combinators, 7
kessel.primitives, 6

Symbols

__call__() (kessel.primitives.Proxy method), 6
__init__() (kessel.primitives.Proxy method), 7
__weakref__ (kessel.primitives.ParseError attribute), 6
__weakref__ (kessel.primitives.ParseError.Entity attribute), 6
__weakref__ (kessel.primitives.Proxy attribute), 7

B

between() (in module kessel.combinators), 7
bind() (kessel.primitives.Proxy method), 7

C

choice() (in module kessel.combinators), 8
count() (in module kessel.combinators), 8

E

error() (in module kessel.primitives), 7

K

kessel.combinators (module), 7
kessel.primitives (module), 6

M

many() (in module kessel.combinators), 8
many1() (in module kessel.combinators), 8
mapf() (in module kessel.combinators), 8
mapf_star() (in module kessel.combinators), 8
match() (in module kessel.primitives), 7

N

not_followed_by() (in module kessel.combinators), 8

O

option() (in module kessel.combinators), 8

P

ParseError, 6
ParseError.Entity (class in kessel.primitives), 6
Proxy (class in kessel.primitives), 6

S

sep_by() (in module kessel.combinators), 9
sep_by1() (in module kessel.combinators), 9
sequence() (in module kessel.combinators), 9
sequence_l() (in module kessel.combinators), 9
sequence_r() (in module kessel.combinators), 9
suppress_expect() (in module kessel.combinators), 9

T

try_() (in module kessel.combinators), 9

U

unit() (in module kessel.primitives), 7
unpeek() (kessel.primitives.ParseError method), 6