Parsetron Documentation

Release 0.1.1

KITT.AI

August 04, 2015

Contents

Abstract

Parsetron is a robust incremental natural language parser utilizing semantic grammars for small focused domains. It is mainly used to convert natural language command to executable API calls (e.g., *"set my bedroom light to red"* –> `set_light('bedroom', [255, 0, 0])`). Parsetron is written in pure Python (2.7), portable (a single `parsetron.py` file), and can be used in conjunction with a speech recognizer.

License Apache License 2.0

Author Xuchen Yao from KITT.AI

Source https://github.com/Kitt-AI/parsetron

Table of Contents

CHAPTER 1

Parsetron Quick Start

1.1 Installation

Parsetron is available through PyPI:

```
pip install parsetron
```

(if you don't have `pip`, follow these instructions to install it)

Alternatively, Parsetron comes as a single `parsetron.py` file. Just download it from the repository and put the file under your `PYTHONPATH` or current directory so that you can do:

```
import parsetron
```

or:

```
from parsetron import *
```

Parsetron can be run with either CPython 2.7 or PyPy. If PyPy is warmed up, the parsing speed is about 3x that of CPython. At the current stage Parsetron doesn't support Python 3 yet.

1.2 Simple Example

```
from parsetron import Set, Regex, Optional, OneOrMore, Grammar, RobustParser

class LightGrammar(Grammar):

    action = Set(['change', 'flash', 'set', 'blink'])
    light = Set(['top', 'middle', 'bottom'])
    color = Regex(r'(red|yellow|blue|orange|purple|...)')
    times = Set(['once', 'twice', 'three times']) | Regex(r'\d+ times')
    one_parse = action + light + Optional(times) + color
    GOAL = OneOrMore(one_parse)

    @staticmethod
    def test():
        parser = RobustParser((LightGrammar()))
        sents = [
            "set my top light to red",
            "set my top light to red and change middle light to yellow",
            "set my top light to red and change middle light to yellow and flash bottom light twice
```

```
        ]
        for sent in sents:
            tree, result = parser.parse(sent)
            assert result.one_parse[0].color == 'red'

            print '"%s"' % sent
            print "parse tree:"
            print tree
            print "parse result:"
            print result
            print
```

output:

```
"set my top light to red"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
      "one_parse": [
        "set",
        "top",
        "red"
      ],
      "color": "red",
      "light": "top"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ]
  ]
}

"set my top light to red and change middle light to yellow"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
  (one_parse
    (action "change")
    (light "middle")
```

```
    (color "yellow")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
      "one_parse": [
        "set",
        "top",
        "red"
      ],
      "color": "red",
      "light": "top"
    },
    {
      "action": "change",
      "one_parse": [
        "change",
        "middle",
        "yellow"
      ],
      "color": "yellow",
      "light": "middle"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ],
    [
      "change",
      "middle",
      "yellow"
    ]
  ]
}

"set my top light to red and change middle light to yellow and flash bottom light twice in blue"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
  (one_parse
    (action "change")
    (light "middle")
    (color "yellow")
  )
  (one_parse
    (action "flash")
    (light "bottom")
```

```
    (Optional(times)
      (times
        (Set(three times|twice|once) "twice")
      )
    )
    (color "blue")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
      "one_parse": [
        "set",
        "top",
        "red"
      ],
      "color": "red",
      "light": "top"
    },
    {
      "action": "change",
      "one_parse": [
        "change",
        "middle",
        "yellow"
      ],
      "color": "yellow",
      "light": "middle"
    },
    {
      "one_parse": [
        "flash",
        "bottom",
        "twice",
        "blue"
      ],
      "color": "blue",
      "Set(three times|twice|once)": "twice",
      "Optional(times)": "twice",
      "times": "twice",
      "light": "bottom",
      "action": "flash"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ],
    [
      "change",
      "middle",
      "yellow"
    ],
```

```
    [
      "flash",
      "bottom",
      "twice",
      "blue"
    ]
  ]
}
```

1.3 Complex Example

```
from parsetron import Set, Regex, Optional, OneOrMore, Grammar, RobustParser


def regex2int(result):
    # result holds Regex(r'\d+ times') lexicon
    num = int(result.get().split()[0])
    result.set(num)


def times2int(result):
    r = result.get().lower()
    mapper = {"once": 1, "twice": 2, "three times": 3}
    num = mapper[r]
    result.set(num)


def color2rgb(result):
    r = result.get().lower()
    # r now holds color lexicons
    mapper = {
        "red": (255, 0, 0),
        "yellow": (255, 255, 0),
        "blue": (0, 0, 255),
        "orange": (255, 165, 0),
        "purple": (128, 0, 128)
    }
    color = mapper[r]
    result.set(color)


class LightAdvancedGrammar(Grammar):

    action = Set(['change', 'flash', 'set', 'blink'])
    light = Set(['top', 'middle', 'bottom'])

    color = Regex(r'(red|yellow|blue|orange|purple|...)').\
        set_result_action(color2rgb)
    times = Set(['once', 'twice', 'three times']).\
        set_result_action(times2int) | \
        Regex(r'\d+ times').set_result_action(regex2int)

    one_parse = action + light + Optional(times) + color
    GOAL = OneOrMore(one_parse)

    @staticmethod
```

```
    def test():
        parser = RobustParser((LightAdvancedGrammar()))
        tree, result = parser.parse("flash my top light twice in red and "
                                    "blink middle light 20 times in yellow")
        print tree
        print result
        assert result.one_parse[0].color == (255, 0, 0)
        assert result.one_parse[0].times == 2
        assert result.one_parse[1].color == (255, 255, 0)
        assert result.one_parse[1].times == 20
        print
```

output:

```
(GOAL
  (one_parse
    (action "flash")
    (light "top")
    (Optional(times)
      (times
        (Set(three times|twice|once) "twice")
      )
    )
    (color "red")
  )
  (one_parse
    (action "blink")
    (light "middle")
    (Optional(times)
      (times
        (Regex(^\d+ times$) "20 times")
      )
    )
    (color "yellow")
  )
)

{
  "one_parse": [
    {
      "one_parse": [
        "flash",
        "top",
        2,
        [
          255,
          0,
          0
        ]
      ],
      "color": [
        255,
        0,
        0
      ],
      "Set(three times|twice|once)": 2,
      "Optional(times)": 2,
      "times": 2,
```

```
      "light": "top",
      "action": "flash"
    },
    {
      "one_parse": [
        "blink",
        "middle",
        20,
        [
          255,
          255,
          0
        ]
      ],
      "Regex(^\\d+ times$)": 20,
      "color": [
        255,
        255,
        0
      ],
      "light": "middle",
      "Optional(times)": 20,
      "times": 20,
      "action": "blink"
    }
  ],
  "GOAL": [
    [
      "flash",
      "top",
      2,
      [
        255,
        0,
        0
      ]
    ],
    [
      "blink",
      "middle",
      20,
      [
        255,
        255,
        0
      ]
    ]
  ]
}
```

1.4 What It is

Parsetron is a semantic parser that converts natural language text into API calls. Typical usage scenarios include for instance:

- control your smart light with language, e.g.:

  - give me something romantic
  - my living room light is too dark
  - change bedroom light to sky blue
  - blink living room light twice in red color
- control your microwave with language, e.g.:
  - defrost this chicken please, the weight is 4 pounds
  - heat up the pizza for 2 minutes 20 seconds
  - warm up the milk for 1 minute

The difficult job here is to extract key information from the natural language command to help developers call certain APIs to control a smart device. Conventional approach is writing a bunch of rules, such as regular expressions, which are difficult to maintain, read, and expand. Computational linguists opt for writing Context Free Grammars. But the learning curve is high and the parser output – usually a constituency tree or a dependency relation – is not directly helpful in our tasks.

Parsetron is designed to tackle these challenges. Its design philosophy is to **make natural language understanding easy** for developers with no background in computational linguistics, or natural language processing (NLP).

Parsetron has the following properties:

- **easy to use**: grammar definition is in Python; thus developers do not have to learn another format (traditionally grammars are usually defined in BNF format).
- **robust**: it omits unknown (not defined in grammar) word when parsing; it also parses multi-token phrases (modern NLP parsers are only single-token based).
- **incremental**: it emits parse result as soon as it's available; this helps in applications requiring quick responding time, such as through speech interaction.
- **flexible**: users can define their own pre-parsing tokenization and post-parsing callback functions in their grammar specification; this assigns developers as much power as Python has.

It understands language per definition of a semantic grammar.

1.5 How it works

Parsetron is a Chart Parser for Context Free Grammars (CFGs). It works in the following way:

1. Accept a grammar extended from *parsetron.Grammar*, which must have a `GOAL` defined (similar to the start symbol `S` in CFGs). The grammar is defined in Python (so no extra learning curve for Python developers!).
2. Tokenize an input string by white spaces.
3. Construct a *parsetron.Chart* and parse with a default Left Corner Top Down strategy.
   - unknown words not defined in the grammar are automatically omitted.
   - if single tokens are not recognized, parsetron also tries to recognize phrases.
4. Output a conventional linguistic tree, and a *parsetron.ParseResult* for easier interpretation.
   - results are ranked by the minimal tree sizes.
   - in the future parsetron will provide probabilistic ranking.
5. Run post processing on the parse result if any call-back functions are defined via the *parsetron.GrammarElement.set_result_action()* function in the grammar.

Parsetron is inspired by pyparsing, providing a lot of classes with the same intuitive names. pyparsing implements a top-down recursive parsing algorithm, which is good for parsing unambiguous input, but not for natural languages. Parsetron is specifically designed for parsing natural language instructions.

CHAPTER 2

Parsetron Tutorial

In the following we provide a simple example of controlling your smart lights with natural language instructions. Corresponding code of this tutorial is hosted on Github.

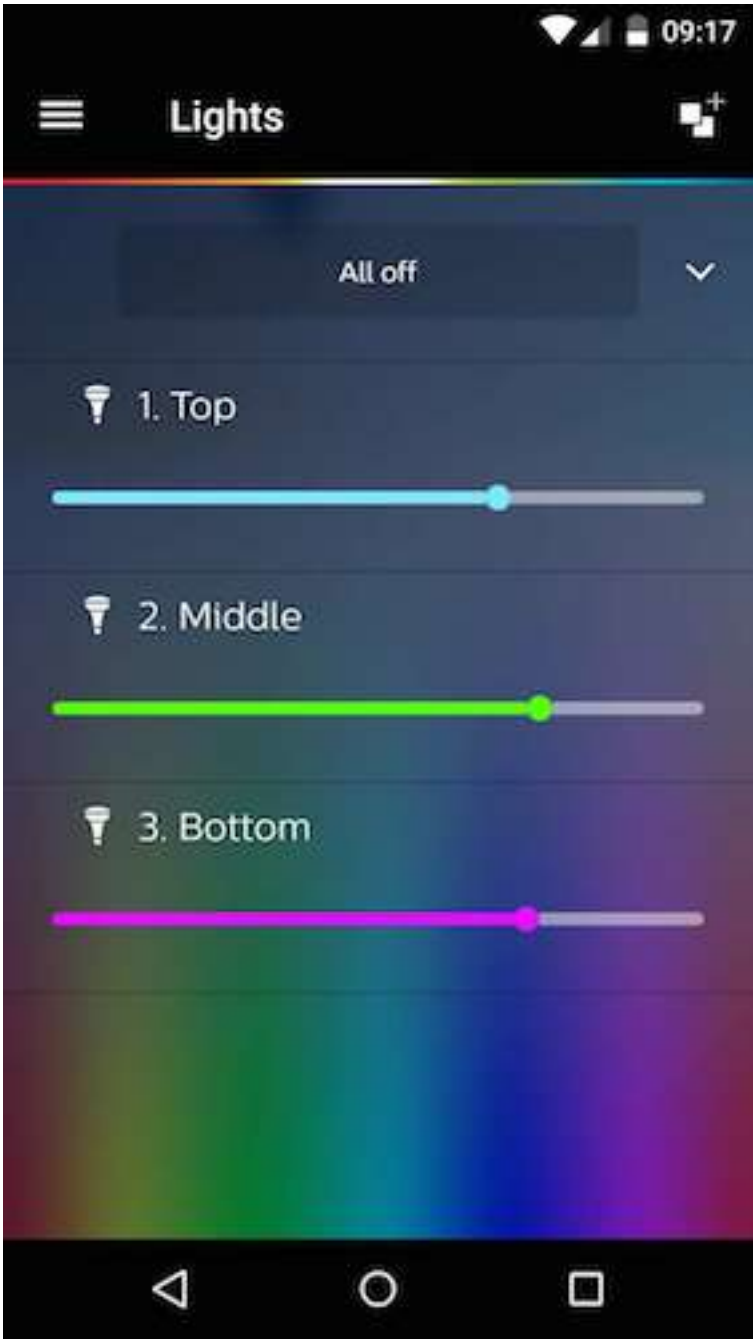
2.1 A Light Grammar

We start by writing a grammar for smart light bulbs. Our lights of interest are the Philips Hue. The starter pack contains three light bulbs and a bridge:

Your application talks to the bridge through shared WiFi and the bridge talks to the light bulbs through Zigbee. The bridge responds to RESTful API calls.

In Kitt.AI office, we have the 3-bulb starter pack set up like this:

The three bulbs are named **top**, **middle**, and **bottom** for easier reference. Most people would name them by *living room*, *bedroom*, etc. It's your choice. However, no matter what you name those bulbs, you **must** register them with the Hue bridge either through their Restful APIs, or through the app:



Now imagine we want to parse the following simple sentences:

1. set my top light to red
2. set my top light to red and change middle light to yellow
3. set my top light to red and change middle light to yellow and flash bottom light twice in blue

We can define a simple grammar in the following Python code:

```
from parsetron import *

class LightGrammar(Grammar):
    action = Set(['change', 'flash', 'set', 'blink'])
```

```
5       light = Set(['top', 'middle', 'bottom'])
6       color = Regex(r'(red|yellow|blue|orange|purple|...)')
7       times = Set(['once', 'twice', 'three times']) | Regex(r'\d+ times')
8
9       one_parse = action + light + Optional(times) + color
10      GOAL = one_parse | one_parse + one_parse | one_parse + one_parse + one_parse
```

The above code defined a minimal grammar that would parse our test sentences. Here's a step-by-step explanation.

- on line 3 we defined a `LightGrammar` class that extends a standard *parsetron.Grammar*. Defining grammars in classes helps modularization.
- on lines 4-5, we used a *parsetron.Set* class to match anything that's in the set:

```
action = Set(['change', 'flash', 'set', 'blink'])
light = Set(['top', 'middle', 'bottom'])
```

- on line 6, instead of using a set, we used a regular expression to encode color names:

```
color = Regex(r'(red|yellow|blue|orange|purple|...)')
```

  Note that there could be hundreds of color names. A *parsetron.Regex* builds a finite state machine to efficiently code them. But of course we can also use a Set.

- on line 7, we introduced the `|` operator, which encodes a *parsetron.Or* relation to specify alternative ways of representing `times`:

```
times = Set(['once', 'twice', 'three times']) | Regex(r'\d+ times')
```

  So `times` can either match *"three times"*, or *"3 times"*.

- on line 9, we defined a `one_parse` of a sentence, which represents a single minimal set of information encoded in a parse:

```
one_parse = action + light + Optional(times) + color
```

  The + operator here encodes a *parsetron.And* relation, matching a sequence of tokens. For unknown words parsetron simply ignores them. The *parsetron.Optional* class is a kind of syntactic sugar indicating that we can match 0 or 1 time of `times` here. Thus this single `one_parse` parses both of the following sentences:

  1. blink my top light in red
  2. blink my top light **twice** in red

  Note that `one_parse` doesn't parse sentences 2 and 3 above, which contain coordination:

  2. **coordination**: set my top light to red **and** change middle light to yellow
  3. **coordination**: set my top light to red **and** change middle light to yellow **and** flash bottom light twice in blue

  thus on line 10 we concatenated `one_parse` two and three times to make parses:

```
GOAL = one_parse | one_parse + one_parse | one_parse + one_parse + one_parse
```

- line 10 is **ugly** however. Alternatively we can write:

```
GOAL = one_parse * [1, 3] # or:
GOAL = one_parse * (1, 3)
```

  meaning that a `GOAL` contains a `one_parse` one to three times. But then it is **not flexible**: what if there's a fourth coordination? So we simply change it to:

```
    GOAL = one_parse * (1, ) # one or more times, but better with:
    GOAL = OneOrMore(one_parse)
```

Now our `GOAL` can parse however many `one_parse` 's using *parsetron.OneOrMore*!

Note: You can freely define all kinds of variables in your grammar, but then **have to** define a `GOAL` so the parser knows where to start. `GOAL` here is equivalent to what conventionally is called the START symbol `S` in CFGs.

Warning: The `|` operator has lower precedence than the + operator. Thus the following code:
`a = b	c + d`
is equal to:
`a = b	(c + d)`
rather than:
`a = (b | c) + d`

Finally we have a very simple grammar defined for smart light:

```
from parsetron import *

class LightGrammar(Grammar):
    action = Set(['change', 'flash', 'set', 'blink'])
    light = Set(['top', 'middle', 'bottom'])
    color = Regex(r'(red|yellow|blue|orange|purple|...)')
    times = Set(['once', 'twice', 'three times']) | Regex(r'\d+ times')

    one_parse = action + light + Optional(times) + color
    GOAL = OneOrMore(one_parse)
```

2.2 Let's Parse It

To parse sentences, we first construct a *parsetron.RobustParser*, then call its *parsetron.RobustParser.parse()* function:

```
parser = RobustParser(LightGrammar()
sents = ["set my top light to red",
    "set my top light to red and change middle light to yellow",
    "set my top light to red and change middle light to yellow and flash bottom light twice in blue"]
for sent in sents:
    tree, result = parser.parse(sent)
    print '"%s"' % sent
    print "parse tree:"
    print tree
    print "parse result:"
    print result
    print
```

And here's the output:

```
"set my top light to red"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
      "one_parse": [
        "set",
        "top",
        "red"
      ],
      "color": "red",
      "light": "top"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ]
  ]
}

"set my top light to red and change middle light to yellow"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
  (one_parse
    (action "change")
    (light "middle")
    (color "yellow")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
      "one_parse": [
        "set",
        "top",
        "red"
      ],
```

```
      "color": "red",
      "light": "top"
    },
    {
      "action": "change",
      "one_parse": [
        "change",
        "middle",
        "yellow"
      ],
      "color": "yellow",
      "light": "middle"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ],
    [
      "change",
      "middle",
      "yellow"
    ]
  ]
}

"set my top light to red and change middle light to yellow and flash bottom light twice in blue"
parse tree:
(GOAL
  (one_parse
    (action "set")
    (light "top")
    (color "red")
  )
  (one_parse
    (action "change")
    (light "middle")
    (color "yellow")
  )
  (one_parse
    (action "flash")
    (light "bottom")
    (Optional(times)
      (times
        (Set(three times|twice|once) "twice")
      )
    )
    (color "blue")
  )
)

parse result:
{
  "one_parse": [
    {
      "action": "set",
```

```
      "one_parse": [
        "set",
        "top",
        "red"
      ],
      "color": "red",
      "light": "top"
    },
    {
      "action": "change",
      "one_parse": [
        "change",
        "middle",
        "yellow"
      ],
      "color": "yellow",
      "light": "middle"
    },
    {
      "one_parse": [
        "flash",
        "bottom",
        "twice",
        "blue"
      ],
      "color": "blue",
      "Set(three times|twice|once)": "twice",
      "Optional(times)": "twice",
      "times": "twice",
      "light": "bottom",
      "action": "flash"
    }
  ],
  "GOAL": [
    [
      "set",
      "top",
      "red"
    ],
    [
      "change",
      "middle",
      "yellow"
    ],
    [
      "flash",
      "bottom",
      "twice",
      "blue"
    ]
  ]
}
```


The *parsetron.RobustParser.parse()* function returns a tuple of (`parse tree`, `parse result`):

1. `parse tree` is a *parsetron.TreeNode* class, mainly for the purpose of eye-checking results.
2. `parse result` is a *parsetron.ParseResult* class. It is converted from `parse tree` and allows intuitive item or attribute setting and getting. For instance:

```
In [7]: result['one_parse']
Out[7]:
[{'action': 'set', 'one_parse': ['set', 'top', 'red'], 'color': 'red', 'light': 'top'},
 {'action': 'change', 'one_parse': ['change', 'middle', 'yellow'], 'color': 'yellow', 'light': '
 {'one_parse': ['flash', 'bottom', 'twice', 'blue'], 'color': 'blue', 'light': 'bottom', 'Option

In [8]: result.one_parse
Out[8]:
[{'action': 'set', 'one_parse': ['set', 'top', 'red'], 'color': 'red', 'light': 'top'},
 {'action': 'change', 'one_parse': ['change', 'middle', 'yellow'], 'color': 'yellow', 'light': '
 {'one_parse': ['flash', 'bottom', 'twice', 'blue'], 'color': 'blue', 'light': 'bottom', 'Option

In [9]: len(result.one_parse)
Out[9]: 3

In [10]: result.one_parse[0].color
Out[10]: 'red'
```

Note here how parsetron has extracted variable names from the `LightGrammar` class to its parse tree and parse result, both explicitly and implicitly. Take the last sentence:

```
{ 'GOAL': [ ['set', 'top', 'red'],
            ['change', 'middle', 'yellow'],
            ['flash', 'bottom', 'twice', 'blue']],
  'one_parse': [ {'action': 'set', 'one_parse': ['set', 'top', 'red'], 'color': 'red', 'light': 'top
                 {'action': 'change', 'one_parse': ['change', 'middle', 'yellow'], 'color': 'yellow',
                 {'one_parse': ['flash', 'bottom', 'twice', 'blue'], 'color': 'blue', 'light': 'botto
```

The implicitly constructed variable names, such as `Optional(times)`, are also present in the result.

The values in parsing results cover the parsed lexicon while respecting the grammar structures. Thus `GOAL` above contains a list of three items, each item is a list of lexical strings itself, corresponding to one `one_parse`.

parsetron also tries to *flatten* the result as much as possible when there is no name conflict. Thus unlike in the parse tree, here `one_parse` is **in parallel** with `GOAL`, instead of **under** GOAL. In this way we can easily access deep items, such as:

```
In [11]: result.one_parse[2].times
Out[11]: 'twice'
```

Otherwise, we would have used something like the following, which is very inconvenient:

```
In [11]: result.GOAL.one_parse[2]['Optional(times)']['times']['Set(Set(three times|twice|once)']
Out[11]: 'twice'
```

2.3 Convert to API Calls

With the parse result in hand, we could easily extract `one_parse`'s from the result and call the Philips Hue APIs. We use the python interface phue for interacting with the hue:

```
# pip install phue
from phue import Bridge

b = Bridge('ip_of_your_bridge')
b.connect()

for one_parse in result.one_parse:
```

```
    if one_parse.action != 'flash':
        b.set_light(one_parse.light, 'xy', color2xy(one_parse.color))
    else:
        # turn on/off a few times according to one_parse.times
```

The above code calls an external function `color2xy()` to convert a string color name to its XY values, which we do not specify here. But more information can be found in core concepts of Hue.

Calling external APIs is beyond scope of this tutorial. But we have a simple working system called firefly for your reference.

2.4 Advanced Usage

So far we have introduced briefly how to parse natural language texts into actions with a minimal grammar for smart lights. But parsetron is capable of doing much more than that, for instance:

- `one_parse.times` is a string (e.g., *"three times"*), we'd like to see instead an integer value (e.g., 3);
- `one_parse.color` is also a string (.e.g., *"red"*, maybe we can directly output its RGB (e.g., (255, 0, 0)) or XY value from the parser too?

In the next page we introduce the *`parsetron.GrammarElement.set_result_action()`* function to post process parse results.

Corresponding code of this tutorial is hosted on Github.

CHAPTER 3

Parsetron Advanced Usage

3.1 Call-back Functions

In the last section we defined `color` and `times` as:

```
color = Regex(r'(red|yellow|blue|orange|purple|...)')
times = Set(['once', 'twice', 'three times']) | Regex(r'\d+ times')
```

A parse result would look something like:

```
{ 'GOAL': [['blink', 'top', 'red', 'twice']],
  'one_parse': [ {'action': 'blink',
                  'one_parse': ['blink', 'top', 'red', 'twice'],
                  'color': 'red',
                  'light': 'top',
                  'times': 'twice'}]}
```

But we'd want something more conveniently like:

```
{ 'GOAL': [['blink', 'top', 'red', 'twice']],
  'one_parse': [ {'action': 'blink',
                  'one_parse': ['blink', 'top', 'red', 'twice'],
                  'color': [255, 0, 0],
                  'light': 'top',
                  'times': 2}]}
```

This can be achieved by the *parsetron.GrammarElement.set_result_action()* call back function, for instance:

```
def color2rgb(result):
    r = result.get().lower()
    # r now holds color lexicons
    mapper = {
        "red": (255, 0, 0),
        "yellow": (255, 255, 0),
        "blue": (0, 0, 255),
        "orange": (255, 165, 0),
        "purple": (128, 0, 128)
    }
    color = mapper[r]
    result.set(color)

color = Regex(r'(red|yellow|blue|orange|purple|...)').set_result_action(color2rgb)
```

The `color2rgb` function now first retrieves the lexicon of color by calling `result.get()` (*parsetron.ParseResult.get()*), then map it to a RGB tuple, and finally replacing the result with `result.set()` (*parsetron.ParseResult.set()*).

Note: The return value of *parsetron.GrammarElement.set_result_action()* is the object itself (`return self`). Thus in the above example `color` is still assigned with the `Regex()` object.

The `times` part is only slightly more complicated as it parses numbers in both digits and words. We define two functions here:

```
def regex2int(result):
    # result holds Regex(r'\d+ times') lexicon
    num = int(result.get().split()[0])
    result.set(num)

def times2int(result):
    r = result.get().lower()
    mapper = {"once": 1, "twice": 2, "three times": 3}
    num = mapper[r]
    result.set(num)

times = Set(['once', 'twice', 'three times']).set_result_action(times2int) | \
    Regex(r'\d+ times').set_result_action(regex2int)
```

Here each grammar element (`Set()` and `Regex()`) has their own call-back functions. Together they define the `times` variable. The result is that the `times` field in parse result is all converted into an integer number, no matter whether it's *twice* or *20 times*.

Next we test whether these call-back functions work as expected!

3.2 Test Your Grammar

The *parsetron.Grammar* class defines a static *parsetron.Grammar.test()* function for testing your grammar. This function is also called by parsetron's pytest routine for both bug spotting and test coverage report. One should freely and fully make use of the `assert` function in `test()` after defining a grammar.

The following is the full grammar with a simple test function:

```
class LightAdvancedGrammar(Grammar):

    action = Set(['change', 'flash', 'set', 'blink'])
    light = Set(['top', 'middle', 'bottom'])

    color = Regex(r'(red|yellow|blue|orange|purple|...)').\
        set_result_action(color2rgb)
    times = Set(['once', 'twice', 'three times']).\
        set_result_action(times2int) | \
        Regex(r'\d+ times').set_result_action(regex2int)

    one_parse = action + light + Optional(times) + color
    GOAL = OneOrMore(one_parse)

    @staticmethod
    def test():
        parser = RobustParser((LightAdvancedGrammar()))
        tree, result = parser.parse("flash my top light twice in red and "
```

```
                                    "blink middle light 20 times in yellow")
        print tree
        print result
        assert result.one_parse[0].color == (255, 0, 0)
        assert result.one_parse[0].times == 2
        assert result.one_parse[1].color == (255, 255, 0)
        assert result.one_parse[1].times == 20
        print
```

Here we make sure that the first `one_parse` structure has its color as the RGB value of red (`result.one_parse[0].color == (255, 0, 0)`) and its times parameter as an integer (`result.one_parse[0].times == 2`). So as the second `one_parse` structure.

Corresponding code of this tutorial is hosted on Github.

Note: When is the call-back function called?

The call-back function is called when we convert the (usually best) parse tree (*parsetron.TreeNode*) to parse result (*parsetron.ParseResult*). It is literally a post-processing function *after* parsing. We cannot call it *during* parsing as a CFG grammar can potentially output many trees while each of these trees might output a different parse result.

3.3 Modularized Grammars

So far we have seen how to convert both colors and numbers into more computer friendly values. However the example code above is too simple to be used in real world. As a matter of fact, both color and number parsing deserve their own grammar. Thus we introduce the notion of **modularized grammar**: each grammar class defines a minimal but fully functional CFG with desired call-back functions; these grammar classes are shared towards bigger and more complex grammars.

We have provided a few examples in the parsetron/grammars folder. For instance, the `parsetron.grammars.NumbersGrammar` in numbers.py parses not only *one/two/three* but even *1 hundred thousand five hundred 61* (100561). The `parsetron.grammars.ColorsGrammar` in colors.py defined over 100 different kinds of colors. All of these definition can be accessed via their `Grammar.GOAL` variable. Then in our lights grammar, we can simply do:

```
from parsetron.grammars.times import TimesGrammar
from parsetron.grammars.colors import ColorsGrammar

class ColoredLightGrammar(Grammar):

    color = ColorsGrammar.GOAL
    times = TimesGrammar.GOAL
    ...
```

In the future we will be adding more grammars as we find useful. If you'd like to contribute your own grammar, send us a pull request! And don't forget to test your grammar (by implementing *parsetron.Grammar.test()*)!

CHAPTER 4

API References

4.1 Class Summary

ParseException	Exception thrown when we can't parse the whole string.
GrammarException	Exception thrown when we can't construct the grammar.
MetaGrammar	A meta grammar used to extract symbol names (expressed as variables) during grammar *cons*
Grammar	Grammar user interface.
GrammarElement	Basic grammar symbols (terminal or non-terminal).
StringCs	Case-sensitive string (usually a terminal) symbol that can be a word or phrase.
String	Case-insensitive version of *StringCs*.
SetCs	Case-sensitive strings in which matching any will lead to parsing success.
Set	Case-insensitive version of *SetCs*.
RegexCs	Case-sensitive string matching with regular expressions.
Regex	Case-insensitive version of *RegexCs*.
GrammarExpression	An expression usually involving a binary combination of two *GrammarElement*'s.
And	An "+" expression that requires matching a sequence.
Or	An "|" expression that requires matching any one.
GrammarElementEnhance	Enhanced grammar symbols for *Optional*/*OneOrMore* etc.
Optional	Optional matching (0 or 1 time).
OneOrMore	OneOrMore matching (1 or more times).
ZeroOrMore	ZeroOrMore matching (0 or more times).
NULL	Null state, used internally
GrammarImpl	Actual grammar implementation that is returned by a *Grammar* construction.
Production	Abstract class for a grammar production in the form:
ExpressionProduction	Wrapper of *GrammarExpression*.
ElementProduction	Wrapper of *GrammarElement*.
ElementEnhanceProduction	Wrapper of *GrammarElementEnhance*.
TreeNode	A tree structure to represent parser output.
Edge	An edge in the chart with the following fields:
Agenda	An agenda for ordering edges that will enter the chart.
ParseResult	Parse result converted from *TreeNode* output, providing easy
Chart	A 2D chart (list) to store graph edges.
IncrementalChart	A 2D chart (list of list) that expands its size as having more edges added.
ChartRule	Rules applied in parsing, such as scan/predict/fundamental.
TopDownInitRule	Initialize the chart when we get started by inserting the goal.
BottomUpScanRule	Rules used in bottom up scanning.
TopDownPredictRule	Predict edge if it's not complete and add it to chart

Table 4.1 – continued fro

LeftCornerPredictScanRule	Left corner rules: only add productions whose left corner non-terminal can parse the lexicon.
BottomUpPredictRule	In bottom up parsing, predict edge if it's not complete and add it to chart
TopDownScanRule	Scan lexicon from top down
CompleteRule	Complete an incomplete edge form the agenda by merging with a matching completed edge f
ParsingStrategy	Parsing strategy used in TopDown, BottomUp, LeftCorner parsing.
TopDownStrategy	Parsing strategy used in TopDown, BottomUp, LeftCorner parsing.
BottomUpStrategy	Parsing strategy used in TopDown, BottomUp, LeftCorner parsing.
LeftCornerStrategy	Parsing strategy used in TopDown, BottomUp, LeftCorner parsing.
RobustParser	A robust, incremental chart parser.

4.2 Class API Details

parsetron.py, a semantic parser written in pure Python.

class parsetron.**Agenda**(**args*, ***kwargs*)
Bases: object

An agenda for ordering edges that will enter the chart. Current implementation is a wrapper around collections.deque.

collections.deque supports both FILO (collections.deque.pop()) and FIFO (collections.deque.popleft()). FILO functions like a stack: edges get immediately popped out after they are pushed in. This has merit of finishing the parse *sooner*, esp. when new edges are just completed, then we can pop them for prediction.

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

append(*edge*)
Add a single *edge* to agenda. *edge* can be either complete or not to be appended to agenda.

Parameters **edge** (Edge) –

extend(*edges*)
Add a sequence of *edges* to agenda.

Parameters **edges** –

Type list(:class'Edge')

pop()
Pop an edge from agenda (stack).

Returns an edge

Return type *Edge*

class parsetron.**And**(*exprs*)
Bases: *parsetron.GrammarExpression*

An "+" expression that requires matching a sequence.

class parsetron.**BottomUpPredictRule**
Bases: *parsetron.ChartRule*

In bottom up parsing, predict edge if it's not complete and add it to chart

class parsetron.**BottomUpScanRule**
Bases: *parsetron.ChartRule*

Rules used in bottom up scanning.

class parsetron.**Chart**(*size*)

Bases: object

A 2D chart (list) to store graph edges. Edges can be accessed via: Chart.edges[start][end] and return value is a set of edges.

Parameters **size** (*int*) – chart size, normally `len(tokens) + 1`.

__weakref__

list of weak references to the object (if defined)

_most_compact_trees(*parent_edge*, *tokens=None*)

Try to eliminate spurious ambiguities by getting the most compact/flat tree. This mainly deals with removing Optional/ZeroOrMore nodes

add_edge(*edge*, *prev_edge*, *child_edge*, *lexicon=u''*)

Add *edge* to the chart with backpointers being *previous edge* and *child edge*

Parameters

- **edge** (Edge) – newly formed edge
- **prev_edge** (Edge) – the left (previous) edge where edge is coming from
- **child_edge** (Edge) – the right (child) edge that the completion of which moved the dot ot prev_edge

Return bool Whether this edge is newly inserted (not already exists)

best_tree_with_parse_result(*trees*)

Return a tuple of the smallest tree among *trees* and its parse result.

Parameters **trees** (*list*) – a list of *TreeNode*

Returns a tuple of (best tree, its parse result)

Return type tuple(*TreeNode*, *ParseResult*)

filter_completed_edges(*start*, *lhs*)

Find all edges with matching *start* position and LHS with *lhs*. directly after the dot as *rhs_after_dot*. For instance, both edges:

```
[1, 1] NP ->  * NNS CC NNS
[1, 3] NP ->  * NNS
```

match `start=1` and `lhs=NP`.

Returns a list of edges

Return type list(*Edge*)

filter_edges_for_completion(*end*, *rhs_after_dot*)

Find all edges with matching `end` position and RHS nonterminal directly after the dot as `rhs_after_dot`. For instance, both edges:

```
[1, 1] NNS ->  * NNS CC NNS
[1, 1] NP ->  * NNS
```

match *end=1* and *rhs_after_dot=NNS*

filter_edges_for_prediction(*end*)

Return a list of edges ending at `end`.

Parameters **end** (*int*) – end position

Returns list(*Edge*)

get_edge_lexical_span(*edge*)
syntactic sugar for calling *self.get_lexical_span(edge.start, edge.end)*

Parameters **edge** (Edge) –

Returns (int, int)

get_lexical_span(*start*, *end=None*)
Get the lexical span chart covers from *start* to *end*. For instance, for the following sentence:

```
please [turn off] the [lights]
```

with parsed items in [], then the lexical span will look like:

```
when end = None, function assumes end=start+1
if start = 0 and end = None, then return (1, 3)
if start = 1 and end = None, then return (4, 5)
if start = 0 and end = 1, then return (1, 5)
```

Parameters

- **start** (*int*) –
- **end** (*int*) –

Returns (int, int)

print_backpointers()
Return a string representing the current state of all backpointers.

set_lexical_span(*start*, *end*, *i=None*)
set the lexical span at *i* in chart to (*start*, *end*). if *i* is None, then default to the last slot in chart (*self.chart_i-1*). *lexical span* here means the span chart at *i* points to in the original sentence. For instance, for the following sentence:

```
please [turn off] the [lights]
```

with parsed items in [], then the lexical span will look like:

```
start = 1, end = 3, i = 0
start = 4, end = 5, i = 1
```

trees(*tokens=None*, *all_trees=False*, *goal=None*)
Yield all possible trees this chart covers. If *all_trees* is False, then only the most compact trees for each *goal* are yielded. Otherwise yield all trees (**warning: can be thousands**).

Parameters

- **tokens** (*list*) – a list of lexicon tokens
- **all_trees** (*bool*) – if False, then only print the smallest tree.
- **goal** – the root of this tree (usually Grammar.GOAL)

Type GrammarElement, None

Returns a tuple of (tree index, TreeNode)

Return type tuple(int, *TreeNode*)

class parsetron.**ChartRule**
Bases: object

Rules applied in parsing, such as scan/predict/fundamental. New rules need to implement the `apply()` method.

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

class parsetron.**CompleteRule**
Bases: *parsetron.ChartRule*

Complete an incomplete edge form the agenda by merging with a matching completed edge from the chart, or complete an incomplete edge from the chart by merging with a matching completed edge from the agenda.

class parsetron.**Edge**(*start*, *end*, *production*, *dot*)
Bases: object

An edge in the chart with the following fields:

Parameters

- **start** (*int*) – the starting position
- **end** (*int*) – the end position, so span = end - start
- **production** (Production) – the grammar production
- **dot** (*int*) – the dot position on the RHS. Any thing before the *dot* has been consumed and after is waiting to complete

get_rhs_after_dot()
Returns the RHS symbol after dot. For instance, for edge:

```
[1, 1] NP ->  * NNS
```

it returns NNS.

If no symbol is after dot, then return None.

Returns RHS after dot

Return type *GrammarElement*

is_complete()
Whether this edge is completed.

Return type bool

merge_and_forward_dot(*edge*)
Move the dot of self forward by one position and change the end position of self edge to end position of *edge*. Then return a new merged Edge. For instance:

```
self: [1, 2] NNS ->  * NNS CC NNS
edge: [2, 3] NNS -> men *
```

Returns a new edge:

```
[1, 3] NNS -> NNS * CC NNS
```

Requires that `edge.start == self.end`

Returns a new edge

Return type *Edge*

scan_after_dot(*phrase*)
Scan *phrase* with RHS after the dot. Returns a tuple of (lexical_progress, rhs_progress) in booleans.

Returns a tuple

Return type tuple(bool, bool)

span()
The span this edge covers, alias of `end - start`. For instance, for edge:

```
[1, 3] NP ->  * NNS
```

it returns 2

Returns an int

Return type int

class parsetron.**ElementEnhanceProduction**(*element*, *rhs=None*)
Bases: *parsetron.ElementProduction*

Wrapper of *GrammarElementEnhance*. An *ElementEnhanceProduction* has the following assertion true:

LHS == RHS[0]

class parsetron.**ElementProduction**(*element*)
Bases: *parsetron.Production*

Wrapper of *GrammarElement*. An *ElementProduction* has the following assertion true:

LHS == RHS[0]

class parsetron.**ExpressionProduction**(*lhs*, *rhs*)
Bases: *parsetron.Production*

Wrapper of *GrammarExpression*.

class parsetron.**Grammar**
Bases: object

Grammar user interface. Users should inherit this grammar and define a final grammar GOAL as class variable.

It's a wrapper around *GrammarImpl* but does not expose any internal functions. So users can freely define their grammar without worrying about name pollution. However, when a *Grammar* is constructed, a *GrammarImpl* is *actually* returned:

```
>>> g = Grammar()
```

now *g* is the real grammar (*GrammarImpl*)

Warning: Grammar elements have to be defined as class variables instead of instance variables for the *Grammar* object to extract variable names in string

Warning: Users have to define a *GOAL* variable in *Grammar* (similar to start variable *S* conventionally used in grammar definition)

__metaclass__
alias of *MetaGrammar*

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

static **test**()
A method to be batch called by pytest (through `test_grammars.py`). Users should give examples of what this Grammar parses and use these examples for testing.

class parsetron.**GrammarElement**
Bases: object

Basic grammar symbols (terminal or non-terminal).

Developers inheriting this class should implement the following functions:

- *_parse()*
- *default_name()*

A grammar element carries the following attributes:

- *is_terminal*: whether this element is terminal or non-terminal. A general rule of thumb is:
  - if it's *GrammarElement*, then terminal;
  - if it's *GrammarExpression*, then non-terminal;
  - if it's *GrammarElementEnhance*, then non-terminal.
- *name*: the name of this element, usually set by the the *set_name()* function or implicitly __call__() function.
- *variable_name*: automatically extracted variable name in string through the *Grammar* construction.
- *canonical_name*: if neither *name* nor *variable_name* is set, then a canonical name is assigned trying to be as expressive as possible.
- *as_list*: whether saves result in a hierarchy as a list, or just flat
- *ignore*: whether to be ignored in ParseResult

__add__(*other*)
Implement the + operator. Returns *And*.

__call__(*name*)
Shortcut for *set_name()*

__mul__(*other*)
Implements the * operator, followed by an integer or a tuple/list:

- `e * m`: `m` repetitions of `e` (`m > 0`)
- `e * (m, n)` or `e * [m, n]`: `m` to `n` repetitions of `e` (all inclusive)
- `m` or `n` in `(m,n)` / `[m,n]` can be None

for instance (=> stands for "is equivalent to"):

- `e * (m, None)` or `e * (m,)` => `m` or more instances of `e` => `e * m` + *ZeroOrMore* `(e)`
- `e * (None, n)` or `e * (0, n)` => `0` to `n` instances of `e`
- `e * (None, None)` => *ZeroOrMore* `(e)`
- `e * (1, None)` => *OneOrMore* `(e)`
- `e * (None, 1)` => *Optional* `(e)`

__or__(*other*)
Implement the | operator. Returns *Or*.

__radd__(*other*)
Implement the + operator. Returns *And*.

__ror__(*other*)
Implement the | operator. Returns *Or*.

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

_parse(*instring*)
Main parsing method to be implemented by developers. Raises ParseException when there is no parse.

Parameters **instring** (*str*) – input string to be parsed

Returns True if full parse else False

Return type bool

Raises *ParseException*

default_name()
default canonical name.

Returns a string

Return type str

ignore()
Call this function to make this grammar element not appear in parse result. :return: self

parse(*instring*)
Main parsing method to be called by users. Raises ParseException when there is no parse. Returns True if the whole string is parsed and False if input string is not parsed but no exception is thrown either (e.g., parsing with Null element)

Parameters **instring** (*str*) – input string

Return bool True if the whole string is parsed else False

production()
converts this GrammarElement (used by User) to a GrammarProduction (used by Parser)

replace_result_with(*value*)
replace the result lexicon with `value`. This is a shortcut to:

```
self.set_result_action(lambda r: r.set(value))
```

Parameters **value** – any object

Returns self

run_post_funcs(*result*)
Run functions set by *set_result_action()* after getting parsing result.

Parameters **result** (ParseResult) – parsing result

Returns None

set_name(*name*)
Set the name of a grammar symbol. Usually the name of a *GrammarElement* is set by its variable name, for instance:

```
>>> light = String("light")
```

but in on-the-fly construction, one can call *set_name()*:

```
>>> Optional(light).set_name("optional_light")
```

or shorten it to a function call like name setting:

```
>>> Optional(light)("optional_light")
```

The function returns a new shallow copied *GrammarElement* object. This allows reuse of common grammar elements in complex grammars without name collision.

Parameters **name** (*str*) – name of this grammar symbol

Returns a self copy (with different id and hash)

Return type *GrammarElement*

set_result_action(**functions*)

Set functions to call after parsing. For instance:

```
>>> number = Regex(r"\d+").set_result_action(lambda x: int(x))
```

It can be a list of functions too:

```
>>> def f1(): pass  # do something
>>> def f2(): pass  # do something
>>> number = Regex(r"\d+").set_result_action(f1, f2)
```

Parameters **functions** – a list of functions

Returns self

yield_productions()

Yield how this element/expression produces grammar productions

class parsetron.**GrammarElementEnhance**(*expr*)

Bases: *parsetron.GrammarElement*

Enhanced grammar symbols for *Optional*/*OneOrMore* etc.

yield_productions()

Yield how this expression produces grammar productions. A *GrammarElementEnhance* class should implement its own.

exception parsetron.**GrammarException**

Bases: exceptions.Exception

Exception thrown when we can't construct the grammar.

__weakref__

list of weak references to the object (if defined)

class parsetron.**GrammarExpression**(*exprs*)

Bases: *parsetron.GrammarElement*

An expression usually involving a binary combination of two *GrammarElement*'s. The resulting GrammarExpression is a non-terminal and does not implement the parsing function _parse().

yield_productions()

Yield how this expression produces grammar productions. A *GrammarExpression* class should implement its own.

class parsetron.**GrammarImpl**(*name*, *dct*)

Bases: object

Actual grammar implementation that is returned by a *Grammar* construction.

__init__(*name*, *dct*)

This *__init__()* function should only be called from *MetaGrammar* but never explicitly.

Parameters

- **name** (*str*) – name of this grammar class
- **dct** (*dict*) – __class__.__dict__ field

__weakref__

list of weak references to the object (if defined)

_build_grammar_recursively(*element*, *productions*)

Build a grammar from *element*. This mainly includes recursively extracting AND/OR *GrammarExpression*'s from *element*.

Parameters

- **element** (GrammarExpression) – a *GrammarExpression*
- **productions** (set) – assign to *set()* when calling the first time

Returns a set of *Production*

Return type set(*Production*)

_eliminate_null_and_expand()

Eliminate the Null elements in grammar by introducing more productions without Null elements. For each production *with* Null, add a new production *without*. For instance:

```
S => Optional(A) B Optional(C)
Optional(A) => NULL    --> remove
Optional(A) => A
Optional(C) => NULL
Optional(C) => C       --> remove
```

becomes:

```
S => Optional(A) B Optional(C)
Optional(A) => A
Optional(C) => C
S => B Optional(C)     --> added
S => Optional(A) B     --> added
S => B                 --> added
```

The rational behind this is that NULL elements call for a lot of extra computation and are highly ambiguous. This function increases the size of grammar but helps gain extra parsing speed. In reality comparison of a parsing task:

- without eliminating: 1.6s, _fundamental_rule() was called 38K times, taking 50% of all computing time. 2c52b18d5fcfb901b55ff0506d75c3f41073871c
- with eliminating: 0.6s, _fundamental_rule() was called 23K times, taking 36% of all computing time. 33a1f3f541657ddf0204d02338d94a7e89473d86

_extract_var_names(*dct*)

Given a dictionary, extract all variable names. For instance, given:

```
light_general_name = Regex(r"(lights|light|lamp)")
```

extract the mapping from `id(light_general_name)` to "light_general_name"

Parameters **dct** (*dict*) – a grammar dictionary

Returns a dictionary mapping from `id(variable)` to variable name.

_set_element_canonical_name(*element*)
Set the canonical name field of *element*, if not set yet

Parameters **element** (GrammarElement) – a grammar element

Returns None

_set_element_variable_name(*element*)
Set the variable_name field of *element*

Parameters **element** (GrammarElement) – a grammar element

Return bool True if found else False

build_leftcorner_table()
For each grammar production, build two mappings from the production to:

1. its left corner RHS element (which is a pre-terminal);
2. the terminal element that does the actual parsing job.

filter_productions_for_prediction_by_lhs(*lhs*)
Yield all productions whose LHS is *lhs*.

Parameters **lhs** (GrammarElement) – a grammar element

Returns a production generator

Return type generator(*Production*)

filter_productions_for_prediction_by_rhs(*rhs_starts_with*)
Yield all productions whose RHS[0] is *rhs_starts_with*.

Parameters **rhs_starts_with** (GrammarElement) – a grammar element

Returns a production generator

Return type generator(*Production*)

filter_terminals_for_scan(*lexicon*)
Yield all terminal productions that parses *lexicon*.

Parameters **lexicon** (*str*) – a string to be parsed

Returns a production generator

Return type generator(*Production*)

get_left_corner_nonterminals(*prod*)
Given a grammar production, return a set with its left-corner non-terminal productions, or a set with *prod* itself if not found, .e.g.:

```
S => A B
A => C D
A => e f
B => b
C => c
```

passing *S* as *prod* will return the set of productions for *A* and *C*.

Parameters **prod** (Production) – a grammar production

Returns set(*Production*)

get_left_corner_terminals(*prod*)
Given a grammar production, return a set with its left-corner terminal productions, or an empty set if not found, .e.g.:

```
S => A B
A => C D
A => e f
B => b
C => c
```

passing *S* as *prod* will return the set of productions for *e* and *c*.

Parameters **prod** (Production) – a grammar production

Returns set(*Production*)

class parsetron.**IncrementalChart**(*init_size=10*, *inc_size=10*)
Bases: *parsetron.Chart*

A 2D chart (list of list) that expands its size as having more edges added.

Parameters

- **size** (*int*) – current size of chart
- **max_size** (*int*) – total capacity of chart, if exceeded, then need to increase by inc_size.
- **inc_size** (*int*) – size to increase when max_size is filled

__init__(*init_size=10*, *inc_size=10*)

Parameters

- **init_size** – the initial size
- **inc_size** – extra size to span when the chart is filled up

increase_capacity()
Increase the capacity of the current chart by *self.inc_size*

class parsetron.**LeftCornerPredictScanRule**
Bases: *parsetron.ChartRule*

Left corner rules: only add productions whose left corner non-terminal can parse the lexicon.

class parsetron.**MetaGrammar**
Bases: type

A meta grammar used to extract symbol names (expressed as variables) during grammar *construction* time. This provides a cleaner way than using obj.__class__.__dict__, whose __dict__ has to be accessed via an extra and explicit function call.

class parsetron.**Null**
Bases: *parsetron.GrammarElement*

Null state, used internally

_parse(*instring*)
Always returns False, no exceptions.

Parameters **instring** (*str*) – input string to be parsed

Returns False

class parsetron.**OneOrMore**(*expr*)
Bases: *parsetron.GrammarElementEnhance*

OneOrMore matching (1 or more times).

yield_productions()
Yield how this expression produces grammar productions. If A = OneOrMore(B), then this yields:

```
A => B
A => B A
```

class parsetron.**Optional**(*expr*)
Bases: *parsetron.GrammarElementEnhance*

Optional matching (0 or 1 time).

yield_productions()
Yield how this expression produces grammar productions. If A = Optional(B), then this yields:

```
A => NULL
A => B
```

class parsetron.**Or**(*exprs*)
Bases: *parsetron.GrammarExpression*

An "|" expression that requires matching any one.

exception parsetron.**ParseException**
Bases: exceptions.Exception

Exception thrown when we can't parse the whole string.

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

class parsetron.**ParseResult**(*name*, *lexicon*, *as_flat=True*, *lex_span=(None*, *None)*)
Bases: object

Parse result converted from *TreeNode* output, providing easy access by list or attribute style, for instance:

```
result['color']
result.color
```

Results are flattened as much as possible, meaning: deep children are elevated to the top as much as possible as long as there are no name conflicts. For instance, given the following parse tree:

```
(GOAL
  (And(action_verb, OneOrMore(one_parse))
    (action_verb "flash")
    (OneOrMore(one_parse)
      (one_parse
        (light_name
          (Optional(light_quantifiers)
            (light_quantifiers "both")
          )
          (ZeroOrMore(light_specific_name)
            (light_specific_name "top")
            (light_specific_name "bottom")
          )
          (Optional(light_general_name)
            (light_general_name "light")
          )
        )
```

```
            (ZeroOrMore(color)
              (color "red")
            )
          )
          (one_parse
            (light_name
              (ZeroOrMore(light_specific_name)
                (light_specific_name "middle")
              )
              (Optional(light_general_name)
                (light_general_name "light")
              )
            )
            (ZeroOrMore(color)
              (color "green")
            )
          )
          (one_parse
            (light_name
              (ZeroOrMore(light_specific_name)
                (light_specific_name "bottom")
              )
            )
            (ZeroOrMore(color)
              (color "purple")
            )
          )
        )
      )
    )
```

The parse result looks like:

```
    {
      "action_verb": "flash",
      "one_parse": [
        {
          "one_parse": "both top bottom light red",
          "light_name": "both top bottom light",
          "light_quantifiers": "both",
          "ZeroOrMore(color)": "red",
          "color": "red",
          "ZeroOrMore(light_specific_name)": "top bottom",
          "Optional(light_general_name)": "light",
          "light_general_name": "light",
          "Optional(light_quantifiers)": "both",
          "light_specific_name": [
            "top",
            "bottom"
          ]
        },
        {
          "one_parse": "middle light green",
          "light_name": "middle light",
          "ZeroOrMore(color)": "green",
          "color": "green",
          "ZeroOrMore(light_specific_name)": "middle",
          "Optional(light_general_name)": "light",
          "light_general_name": "light",
```

```
        "light_specific_name": "middle"
      },
      {
        "one_parse": "bottom purple",
        "light_name": "bottom",
        "ZeroOrMore(color)": "purple",
        "color": "purple",
        "ZeroOrMore(light_specific_name)": "bottom",
        "light_specific_name": "bottom"
      }
    ],
    "And(action_verb, OneOrMore(one_parse))": "flash both top bottom
    light red middle light green bottom purple",
    "GOAL": "flash both top bottom light red middle light green bottom
    purple",
    "OneOrMore(one_parse)": "both top bottom light red middle light green
    bottom purple"
  }
```

The following holds true given the above result:

```
assert result.action_verb == "flash"
assert result['action_verb'] == "flash"
assert type(result.one_parse) is list
assert result.one_parse[0].color == 'red'
assert result.one_parse[0].light_specific_name == ['top', 'bottom']
assert result.one_parse[1].light_specific_name == 'middle'
```

Note how the parse result is flattened w.r.t. the tree. Basic principles of flattening are:

- value of result access is either a string or another *ParseResult* object
- If a node has >= 1 children with the same name, make the name hold a list
- Else make the name hold a string value.

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

add_item(*k*, *v*)
Add a `k => v` pair to result

add_result(*result*, *as_flat*)
Add another result to the current result.

Parameters

- **result** (ParseResult) – another result
- **as_flat** (*bool*) – whether to flatten *result*.

get(*item=None*, *default=None*)
Get the value of `item`, if not found, return `default`. If `item` is not set, then get the main value of ParseResult. The usual value is a lexicon string. But it can be different if the *ParseResult.set()* function is called.

items()
Return the dictionary of items in result

keys()
Return the set of names in result

lex_span(*name=None*)
Return the lexical span of this result (if *name=None*) or the result *name*. For instance:

```
_, result = parser.parse('blink lights')
assert result.lex_span() == (1,3)
assert result.lex_span("action") == (1,2)
```

Parameters **name** (*str*) – when None, return self span, otherwise, return child result's span

Returns (int, int)

name()
Return the result name

Returns a string

Return type str

names()
Return the set of names in result

set(*value*)
Set the value of ParseResult. `value` is not necessarily a string though: post functions from *`GrammarElement.set_result_action()`* can pass a different value to `value`.

values()
Return the set of values in result

class parsetron.**ParsingStrategy**(*rule_list*)
Bases: object

Parsing strategy used in TopDown, BottomUp, LeftCorner parsing. Each strategy consists of a list of various `ChartRules`'s.

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

class parsetron.**Production**(*lhs*, *rhs*)
Bases: object

Abstract class for a grammar production in the form:

LHS –> RHS (RHS is a list)

A grammar **production** is used by the parser while a grammar **element** by the user.

Parameters

- **lhs** – a single LHS of *`GrammarElement`*
- **rhs** (*list*) – a list of RHS element, each of which is of *`GrammarElement`*

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

static factory(*lhs*, *rhs=None*)
a Production factory that constructs new productions according to the type of *lhs*. Users can either call this function, or directly call Production constructors.

Parameters

- **lhs** – a single LHS of *`GrammarElement`*
- **rhs** (list, *`GrammarElement`*) – RHS elements (single or a list), each of which is of *`GrammarElement`*

class parsetron.**Regex**(*pattern*, *flags=2*, *match_whole=True*)
Bases: *parsetron.RegexCs*

Case-insensitive version of *RegexCs*.

class parsetron.**RegexCs**(*pattern*, *flags=0*, *match_whole=True*)
Bases: *parsetron.GrammarElement*

Case-sensitive string matching with regular expressions. e.g.:

```
>>> color = RegexCs(r"(red|blue|orange)")
>>> digits = RegexCs(r"\d+")
```

Or pass a compile regex:

```
>>> import re
>>> color = RegexCs(re.compile(r"(red|blue|orange|a long list)"))
```

Parameters

- **flags** (*int*) – standard re flags
- **match_whole** (*bool*) – whether matching the whole string (default: True).

Warning: if match_whole=False, then r"(week|weeks)" will throw a *ParseException* when parsing "weeks", but r"(weeks|week)" will succeed to parse "weeks"

RegexType
alias of SRE_Pattern

class parsetron.**RobustParser**(*grammar*, *strategy=<parsetron.ParsingStrategy object>*)
Bases: object

A robust, incremental chart parser.

Parameters

- **grammar** – user defined grammar, a *GrammarImpl* type.
- **strategy** (ParsingStrategy) – top-down or bottom-up parsing

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

_parse_multi_token(*sent_or_tokens*, *chart=None*, *lex_start=None*)
Parse sentences while being able to tokenize multiple tokens, for instance:

kill lights -> "kill" "lights" turn off lights -> "turn off" "lights"

Each quotes-enclosed (multi-)token is recognized as a phrase.

This function doesn't parse unrecognizable tokens.

clear_cache()
Clear all history when the parser is to parse another sentence. Mainly used in server mode for incremental parsing

incremental_parse(*single_token*, *is_final*, *only_goal=True*, *is_first=False*)
Incremental parsing one token each time. Returns the best parsing tree and parse result.

Parameters

- **single_token** (*str*) – a single word

- **is_final** (*bool*) – whether the current *single_token* is the last one in sentence.
- **only_goal** (*bool*) – only output trees with GOAL as root node
- **is_first** (*bool*) – whether *single_token* is the first token

Returns (best parse tree, parse result)

Return type tuple(*TreeNode*, *ParseResult*) or (None, None)

incremental_parse_to_chart(*single_token*, *chart*)
Incremental parsing one token each time. Returns (chart, is_token_accepted).

Parameters

- **single_token** (*str*) – a single word
- **chart** (*RobustChart*) – the previous returned chart. On first call, set it to None.

Returns a tuple of (chart, parsed_tokens)

parse(*string*)
Parse an input sentence in `string` and return the best (tree, result).

Parameters **string** – tokenized input

Returns (best tree, best parse)

Return type tuple(*TreeNode*, *ParseResult*)

parse_multi_token_skip_reuse_chart(*sent*)
Parse sentence with capabilities to:

- ***multi_token*: recognize multiple tokens as one phrase** (e.g., "turn off")
- ***skip*: throw away tokens not licensed by grammar (e.g.,** speech fillers "um...")
- ***reuse_chart*: doesn't waste computation by reusing the chart** from last time. This makes the function call up to 25% faster than without reusing the chart.

Parameters **sent** (*str*) – a sentence in string

Returns the chart and the newly parsed tokens

Return type tuple(*Chart*, list(str))

parse_string(*string*)
alias of *parse()*.

parse_to_chart(*string*)
Parse a whole sentence into a chart and parsed tokens. This gives a raw chart where all trees or the single best tree can be drawn from.

Parameters **string** (*str*) – input sentence that's already tokenized.

Returns parsing chart and newly parsed tokens

Return type tuple(*Chart*, list(str))

print_parse(*string*, *all_trees=False*, *only_goal=True*, *best_parse=True*, *print_json=False*, *strict_match=False*)
Print the parse tree given input `string`.

Parameters

- **string** (*str*) – input string
- **all_trees** (*bool*) – whether to print all trees (warning: massive output)

- **only_goal** (*bool*) – only print the tree licensed by final goal
- **best_parse** (*bool*) – print the best one tree ranked by the smallest size
- **strict_match** (*bool*) – strictly matching input with parse output (for test purposes)

Returns True if there is a parse else False

class parsetron.**Set**(*strings*)
Bases: *parsetron.SetCs*

Case-insensitive version of *SetCs*.

class parsetron.**SetCs**(*strings*, *caseless=False*)
Bases: *parsetron.GrammarElement*

Case-sensitive strings in which matching any will lead to parsing success. This is a short cut for disjunction of *StringCs* s (|), or *Regex* (`r'(a\|b\|c\|...)'`).

Input can be one of the following forms:

•a string with elements separated by spaces (defined by regex `r"\s+"`)

•otherwise an iterable

For instance, the following input is equivalent:

```
>>> "aa bb cc"
>>> ["aa", "bb", "cc"]
>>> ("aa", "bb", "cc")
>>> {"aa", "bb", "cc"}
```

The following is also equivalent:

```
>>> "0123...9"
>>> "0 1 2 3 .. 9"
>>> ["0", "1", "2", "3", ..., "9"]
>>> ("0", "1", "2", "3", ..., "9")
>>> {"0", "1", "2", "3", ..., "9"}
```

class parsetron.**String**(*string*)
Bases: *parsetron.StringCs*

Case-insensitive version of *StringCs*.

class parsetron.**StringCs**(*string*)
Bases: *parsetron.GrammarElement*

Case-sensitive string (usually a terminal) symbol that can be a word or phrase.

class parsetron.**TopDownInitRule**
Bases: *parsetron.ChartRule*

Initialize the chart when we get started by inserting the goal.

class parsetron.**TopDownPredictRule**
Bases: *parsetron.ChartRule*

Predict edge if it's not complete and add it to chart

class parsetron.**TopDownScanRule**
Bases: *parsetron.ChartRule*

Scan lexicon from top down

class parsetron.**TreeNode**(*parent*, *children*, *lexicon*, *lex_span*)

Bases: object

A tree structure to represent parser output. parent should be a chart *Edge* while children should be a *TreeNode*. *lexicon* is the matched string if this node is a leaf node.

Parameters

- **parent** (Edge) – an edge in Chart
- **children** (*list*) – a list of *TreeNode*
- **lexicon** (*str*) – matched string when this node is a leaf node.
- **lex_span** ((*int,int*)) – (start, end) of lexical token offset.

__weakref__

list of weak references to the object (if defined)

dict_for_js()

represents this tree in dict so a json format can be extracted by:

```
json.dumps(node.dict_for_js())
```

Returns a dict

size()

size is the total number of non-terminals and terminals in the tree

Returns int

Return type int

to_parse_result()

Convert this TreeNode to a ParseResult. The result is flattened as much as possible following:

- if the parent node has as_list=True (ZeroOrMore and OneOrMore), then its children are not flattened;
- children are flattened (meaning: they are elevated to the same level as their parents) in the following cases:
  - child is a leaf node
  - parent has as_list=False **and** all children have no name conflicts (e.g., in p -> {c1 -> {n -> "lexicon1"}, c2 -> {n -> "lexicon2"}}, n will be elevated to the same levels of c1 and c2 separately, but not to the same level of p).

Returns *ParseResult*

class parsetron.**ZeroOrMore**(*expr*)

Bases: *parsetron.GrammarElementEnhance*

ZeroOrMore matching (0 or more times).

yield_productions()

Yield how this expression produces grammar productions. If A = ZeroOrMore(B), then this yields:

```
A => NULL
A => B
A => B A
```

or (semantically equivalent):

```
        A => NULL
        A => OneOrMore(B)
```

parsetron.**_ustr**(*obj*)

Drop-in replacement for str(obj) that tries to be Unicode friendly. It first tries str(obj). If that fails with a UnicodeEncodeError, then it tries unicode(obj). It then < returns the unicode object | encodes it with the default encoding | ... >.

parsetron.**find_word_boundaries**(*string*)

Given a string, such as "my lights are off", return a tuple:

```
    0: a list containing all word boundaries in tuples
    (start(inclusive), end(exclusive)):
    [(0, 2), (3, 9), (10, 13), (14, 17)]
    1: a set of all start positions: set[(0, 3, 10, 14)]
    2: a set of all end positions: set[(2, 9, 13, 17)]
```

parsetron.**strip_string**(*string*)

Merge spaces into single space

p

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

V

Y

Z