# EBNFParser Documentation

*Release 2.0*

**thautwarm**

**Apr 05, 2018**

# Notes

CHAPTER 1

# Quick Start

## 1.1 Installing

The EBNFParser only supports Python 3.6+ now.

You can install it by using **PyPI**.

```
pip install -U EBNFParser
```

## 1.2 Hello World

We can try to parse Lisp grammar syntax into AST(Abstract Synatx Tree) as our first attempt.

```
(define add3 (x y z)
          (add x
              (add y z)))
```

Here is a source code example:

- lisp.ruiko

```
ignore [space]  # ignore the tokens with this(these) name(s).

space   := R'\s';

Atom    := R'[^\(\)\s\`]+'; # use Regex

Expr    ::= Atom
        | Quote
        | '(' Expr* ')';


Quote   ::=  '`' Expr ;
```

```
Stmts   ::= Expr*;
```

And then use it to generate a parser and make a test script automatically by EBNFParser.

Finally, test it.

```
ruiko lisp.ruiko lisp_parser.py --test
python test_lang.py Stmts "(definie f (x y z) (add (add x y) z))"
 ==========================ebnfparser test script================================
Stmts[
    Expr[
        [name: auto_const, string: "("]
        Expr[
            [name: Atom, string: "definie"]
...(omit)
```

## 1.3 Integrate EBNFParser Into Your Own Project

For example, if we have generated the lisp parser file like the above as a module `MyProject.Lisp.parser`.

```
from Ruikowa.ObjectRegex.ASTDef import Ast
from Ruikowa.ErrorHandler import ErrorHandler
from Ruikowa.ObjectRegex.MetaInfo import MetaInfo
from Ruikowa.ObjectRegex.Tokenizer import Tokenizer

from lisp_parser import Stmts, token_table

import typing as t

def token_func(src_code: str) -> t.Iterable[Tokenizer]:
    return Tokenizer.from_raw_strings(
        src_code, token_table, ({"space"}, {}))

parser = ErrorHandler(Stmts.match, token_func)

def parse(filename: str) -> Ast:

    return parser.from_file(filename)

# just create a file `test.lisp` and write some lisp codes.
print(parse("./test.lisp"))
```

An `Ruikowa.ObjectRegex.Ast` is a nested list of Tokenizers, for instance:

```
AstName[
    AstName[
        Tokenizer1
        Tokenizer2
        AstName[
            ...
        ]
    ]
    Tokenizer3
]
```

You can use `obj.name` to get the name of an instance of `Ast` or `Tokenizer`.

CHAPTER 2

# Parsing in EBNFParser

EBNFParser is a parser generator framework to parse raw string into structured nested list(AST).

Pasring of EBNFParser has following steps:

## 2.1 Tokenizing

Tokenizing is the very first step to split input string into a sequence of `Ruikowa.ObjectRegex.Tokenizer` objects.

A `Ruikowa.ObjectRegex.Tokenizer` has the following **readonly** attributes:

- **name** [str] type of the tokenizer.
- **string** [str] string content(from input raw string) of the tokenizer.
- **colno** [int] column number in current file.
- **lineno** [int] row number in current file.

Example:

- parsing_tokenizing.ruiko

```
MyTokenType  := 'abc' '233';
# The above syntax defines a literal parser to parse strings like "abc" or "233".
# "abc", "233" will be added into `token_table` to generate automatical tokenizing␣
↪function.

parserToTest ::= MyTokenType+;
# The above syntax defines a combined parser with `MyTokenType`.

# A combined parser is combined by several literal parsers and other combined parsers,
#   which can handle very complicated cases of a sequence of `Ruikowa.ObjectRegex.
↪Tokenizer` objects.
```

- compile it

```
ruiko parsing_tokenizing.ruiko parsing_tokenizing.py --test
```

- test it

```
python test_lang.py parserToTest "abc233233"
=========================ebnfparser test script================================
parserToTest[
    [name: MyTokenType, string: "abc"]
    [name: MyTokenType, string: "233"]
    [name: MyTokenType, string: "233"]
]
```

Take care that if you're using anonymous literal pattern when definining a combined parser, like the following:

```
Just ::= 'just'+;
```

Then name of all the anonymous tokenizers is just `"auto_const"` :

```
ruiko just.ruiko just --test
python test_lang.py Just "justjustjust"
=========================ebnfparser test script================================
Just[
    [name: auto_const, string: "just"]
    [name: auto_const, string: "just"]
    [name: auto_const, string: "just"]
]
```

## 2.2 CastMap(Optional)

Sometimes we need special cases, a vivid instance is `keyword` .

The string content of a `keyword` could be also matched by `identifier` (in most programming languages we have identifiers), just as the following case:

- parsing_CastMap.ruiko

```
ignore [space]
space         := R'\s+';
# ignore the whitespace characters.


identifier    := R'[a-zA-Z_]{1}[a-zA-Z_0-9]*';
keyword       := 'def' 'for' 'public';

parserToTest ::= (identifier | keyword)+;
```

There is no doubt that `identifier` will cover the cases of `keyword`

```
ruiko parsing_CastMap.ruiko parsing_CastMap.py --test
python test.py parserToTest "def for public"
=========================ebnfparser test script================================
parserToTest[
    [name: identifier, string: "def"]
    [name: identifier, string: "for"]
    [name: identifier, string: "public"]
]
```

Take care that all of the Tokenizers have name **identifier**, not **keyword** ! As as result, the keyword could be used in some illegal places, just like:

```
for = 1
for for <- [for] do
    for
```

The above example might not trouble you, but of course there could be something severer.

I'd like to give a solution adopted by EBNFParser auto-token.

(modify parsing_CastMap.ruiko

```
identifier   := R'[a-zA-Z_]{1}[a-zA-Z_0-9]*';
keyword cast := 'def' 'for' 'public';
```

Here we define a `cast map` that will map the string tokenized by `identifier`(like :code:"def", *:code:*"for"' and `"public"`) to a **const string**, and output a `Ruikowa.ObjectRegex.Tokenizer` which name is a **const string** `"keyword"`.

```
ruiko parsing_CastMap.ruiko parsing_CastMap.py --test
python test.py parserToTest "def for public other"
=========================ebnfparser test script================================
 parserToTest[
    [name: keyword, string: "def"]
    [name: keyword, string: "for"]
    [name: keyword, string: "public"]
    [name: identifier, string: "other"]
]
```

Perfect!

## 2.3 ReStructure Tokenizers

This is what the word "parsing" accurately means.

Maybe you've heard about some sequence operation like `flatMap` (Scala-flatMap) , `collect` (FSharp-collect) , `selectMany` (Linq-SelectMany), that's great, because parsing is its inverse!

```
raw words :

    ["def", "f", "(", "x", ")", "=", "x"]

after parsing there is an AST:

    FunctionDef[
        "f"
              # "def" is thrown away because it's useless to semantics, but you can
              # preserve it, causing noises. The same below.
        ArgList[
            "x"
        ],

        Expression[
            "x"
        ]
    ]
```

And structures of the parsed just match what you defined with EBNF.

Here is an example to generate above AST by using a EBNF idiom - `ruiko` which is proposed by EBNFParser to extend primary EBNF.

```
keyword     cast as K       := 'def';
identifier                  := R'[a-zA-Z_]{1}[a-zA-Z_0-9]*';
FunctionDef throw ['def']   ::= K'def' identifier '(' ArgList ')' '=' Expression;
Expression                  ::= ... # omit
ArgList                     ::= ... # omit
```

What's more, EBNFParser supports unlimited **left recursions**.

CHAPTER 3

# Ruiko EBNF

## 3.1 Grammar

```
ignore [token1, token2, ...]
# optional, discard some tokenizers with specific names.
# it only affects when you're using EBNFParser automatical tokenizing function.

deftoken directory1.directory2...directoryn.filename
# your custom token function. cannot be applied when you're using auto token.


token1  := ...;
token2  := ...;

token3 cast := ...;
# define a cast map

token4 cast as K := ...;
# def cast map and custom prefix

token5 as K := ...;
# only def custom prefix


token6 := ...;

token7 of token5 := ...;
# add more patterns to token5

parser1 ::=  token3 token5+ | [token6] token4* (parser2 parser3){3, 10};
# define a combined parser
/*
   `|` means `or`,
   `[<patterns>]` means `optional`,
```

```
   `(<patterns>)` means `make a new pattern by several patterns`,
   `pattern+` means one or more,
   `pattern*` means zero or more,
   `pattern{a, b}` means matching this pattern more than `a` times and less than b;
   `pattern{a}` means matching this pattern more than `a` times.
*/

parser2 throw [parser3 ';'] = parser3 parser1 ';';
/*
the result from `parser2` will not contains
    a term(`Tokenizer` or `Ast`) with name=parser3 or string=";"
*/
```

More accurately, see the bootstrap grammar here.

## 3.2 Regex Prefix

Regex prefix in ruiko EBNF would add a regex pattern to `token_table`, which might be used for generating an automatical tokenizing function(unless you use your custom tokenizing function).

When you want to use Regex prefix, just type `R'<your regex pattern>'`.

- url.ruiko

```
url := R'https.*?\.(com|cn|org|net)';
other := R'.';
parserToTest throw [other] ::= (url | other)+;
```

test it .. code

```
ruiko url.ruiko url --test
python test_lang.py parserToTest "https://github.comasdas https://123.net"
=========================ebnfparser test script================================
parserToTest[
    [name: url, string: "https://github.com"]
    [name: url, string: "https://123.net"]
]
```

You should take care that there is only regex matching in tokenizing process, and when literal parsers and combined parsers are parsing tokenizers, they are matching whether the name is what they expect(in fact, what parsers are comparing by is not the **name**, it's the **memory address**, so EBNFParser is very quick in this process).

## 3.3 Cast Map

```
SomeToken cast as S := 'abc';
Alpha               := R'[a-z]+';
F                   ::= S'abc' | Alpha;
```

The ruiko codes above defines a tokenize named `SomeToken` with a prefix `S`.

When the input source is splitted into a sequence of tokenizers , however, even the literal parser `Alpha` is supposed to match all string matched by regex pattern `"[a-z]+"`, it cannot match a tokenizer with attribute `string="abc"` generated by EBNFParser automatical tokenizing, that's because all the `"all"` has been casted into a unique string in a buffer pool, and **all of them have the same name** `SomeToken`, **not** `Alpha`.

Here is a string with value `"abc"` located at an unique memory address, and every literal parser defined by `"abc"` just matched it only.

Just as what I told you at Section `Regex Prefix` , The literal parser defined as `Alpha := R'[a-z]+'` just matches the tokenizer whose name is `Alpha`.

## 3.4 Custom Prefix

If you're using custom tokenizing, several `Ruikowa.ObjectRegex.Tokenizer` objects with the same attribute `string="abc"` (and have the same memory address) could have different names.

To distinguish from each other, you can do as the following:

- Grammar

```
SomeToken as S := 'abc';
Alpha          := R'[a-z]+';
F              ::= S'abc' | Alpha;
G              ::= 'abc';
H              ::= G | F ;
```

```
[name: SomeToken, string: "abc"]
...
```

If you are using combined parser `G` to match above tokenizers, you'll fail, because in the grammar `G` is defined as `G::='abc'` , it means `G` only accepts the a tokenizer who has an attribute `name="auto_const"` and another attribute `string="abc"` (and it's from the unique buff pool, not a string created by regex matching).