PYLERI 1.2.2

Воробій В.Д. КІ-31 https://github.com/transceptor-technology/pyleri

ЗМІСТ

ВСТУП

Pyleri - це парсер, що спеціально створений для SiriDB - бази даних з відкритим програмним кодом, що характеризується надзвичайною швидкодією і добре підходить для хмарних рішень. До того в цій базі даних для парсингу використовувався модуль *lparsing*.

Проте в ньому виявили помилки і він був не зручний для підтримки мови у всіх проектах. Саме тому розробники Pyleri надали можливість експортувати граматику в мови програмування C, Python, Java, Go і Javascript.

1. ВСТАНОВЛЕННЯ ПАКУНКУ

Для встановлення пакунку Pyleri необхідно перш за все встановити систему керування пакунків pip. Потрібно в консоль ввести наведені нижче команди:

```
curl https   bootstrap.pypa.io get-pip.py  o get-pip.py

python get-pip.py
```

цього можна встановити і сам пакунок:

```
sudo pip3 install pyleri
```

Цей пакунок не містить жодних залежностей, тому всі програми що будуть наведені нижче повинні працювати.

2. ПЕРША ПРОГРАМА

Для початку створимо граматику для співставлення стрічки, що починається з ключового слова hi і через пробіл очікує стрічку довільної довжини. Код програми наведений на Рис 3.1.

```
from pyleri import (
    Grammar,
    Keyword,
    Regex,
    Sequence)


# Потрібно успадкувати клас Grammar для визначення мови
class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    START = Sequence(k_hi, r_name)


# Граматика автоматично скомпільовується при створені екземпляру класу
my_grammar = MyGrammar()

# Використання екземлляру класу для тестування на різних стрічках
print(my_grammar.parse('hi "Iris"').is_valid)
print(my_grammar.parse('bye "Iris"').is_valid)
print(my_grammar.parse('bye "Iris"').as_str())
```

Рисунок 3.1

Перш за все, ми імпортуємо необхідні нам класи з модуля Pyleri. Для створення граматики нам необхідно створити клас, що успадковується від класу Grammar, що якраз повинен бути імпортований. В створеному нами класі потрібно як мінімум перевизначити поле START, щоб парсер знав де починати парсинг.

У нашому випадку, START являється послідовністю (sequence) з ключового слова (keyword) і регулярного виразу.

Тож при створенні екземпляру нашого класу граматика автоматично скомпільовується і цей об’єкт містить метод *parse*, що може перевірити довільну стрічку на співставлення з заданою граматикою.

Метод *parse* в свою чергу повертає об’єкт типу Result, що надає зручний інтерфейс, для інформування користувача на рахунок помилки.

Запустимо нашу першу програму, ввівши команду в консоль:

```
python3 example_1.py
```

І отримуємо вивід у консоль:

```
True
False
error at position 0, expecting: hi
```

Рисунок 3.2 Результат роботи першої програми

Тож, як бачимо згідно Рис 3.2 метод *is_valid* повертає булеве значення, що інформує нас щодо правильності стрічки. В свою чергу метод *as_str* надає нам повідомлення в чому полягає помилка.

3. ГРАМАТИКА

Ми вже коротко познайомились з класом Grammar у попередньому розділі, проте він ще містить деякі поля, що ми можем переозначити, як наприклад RE_KEYWORDS. Це ми розглянемо пізніше.

Покажемо найцікавіше - як експортувати граматику в інші мови програмування.

Для цього цей клас клас містить такі методи як:

1. *export_js*
2. *export_c*
3. *export_py*
4. *export_go*
5. *export_java*

3.1. ЕКСПОРТ ГРАМАТИКИ В МОВУ ПРОГРАМУВАННЯ JAVASCRIPT

Видозмінимо трохи програмув наведену вище, експортуючи граматику в код мовою програмування Javascript.

Для цього застосуємо функцію *export_js*, що приймає три необов’язкові аргументи:

1. **js_module_name** : Назва Javascript модулю (за замовчуванням ‘jsleri’)
2. **js_template** : Шаблон стрічки, що використовується для експорту (за замовчуванням *Grammar.JS_TEMPLATE*)
3. **js_indent** : відступи, що застосовуються в Javascript файлі (за замовчуванням 4 пробіли)

```
from pyleri import (
    Grammar,
    Keyword,
    Regex,
    Sequence)


class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    START = Sequence(k_hi, r_name)


my_grammar = MyGrammar()

js_str = MyGrammar().export_js(
    js_module_name='jsleri',
    js_template=Grammar.JS_ES6_IMPORT_EXPORT_TEMPLATE,
    js_indent=' ' * 4)

print(js_str)
```

Отже, код програми практично залишився незмінним. В цьому прикладі ми експортуємо граматику в код програми на Javascript, вказуючи шаблон, за яким буде він буде генеруватись. В цьому випадку я вказав генерувати код відповідно до стандарту ES6, що зараз є провідним.
І перенаправляємо його на стандартний вивід (STDOUT).

Запустимо нашу програму, перенаправляючи STDOUT в файл *generatedES6.js* для зручності:

```
python3 example_2.py >> generatedES6.js
```

Вміст файлу *generatedES6.js* після виконання програми:

```
/*
 * This grammar is generated using the Grammar.export_js() method and
 * should be used with the `jsleri` JavaScript module.
 *
 * Source class: MyGrammar
 * Created at: 2020-03-31 14:23:05
 */

import { Regex, Sequence, Grammar, Keyword } from 'jsleri';

class MyGrammar extends Grammar {
    static r_name = Regex('^(?:"(?:[^"]*)")+');
    static k_hi = Keyword('hi');
    static START = Sequence(
        MyGrammar.k_hi,
        MyGrammar.r_name
    );

    constructor() {
        super(MyGrammar.START, '^\\w+');
    }
}



export default MyGrammar;
```

3.2. ЕКСПОРТ ГРАМАТИКИ В МОВУ ПРОГРАМУВАННЯ C

Аналогічно цей модуль підтримує експорт граматики в код на мові програмування C.

Застосуємо функцію *export_c* , що приймає два необов’язкові аргументи:

1. **target** : назва модулю (за замовчуванням 'grammar')
2. **c_indent** : відступ, що застосовується у файлі (за замовчуванням 4 пробіли)

В цьому випадку функція повертає кортеж, що містить код файлу-заголовку (header) і код реалізації (source).

```
from pyleri import (
    Grammar,
    Keyword,
    Regex,
    Sequence)


class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    START = Sequence(k_hi, r_name)


file_name = input("Введіть назву файла : ")  # Назва файлу для виводу

file_name = file_name.strip()  # Видаляємо зайві пробіли зліва і справа

my_grammar = MyGrammar()

code = my_grammar.export_c(
    target=file_name,
    c_indent=" " * 4
)
```

```
source = code[0]  # Програмний код
header = code[1]  # Код файлу-заголовку

# Занесення вмісту програмного коду і коду заголовка у відповідні
# файли з розширенням .h і .c

f = open(file_name + ".c", "w")
f.write(source)
f.close()

f = open(file_name + ".h", "w")
f.write(header)
f.close()
```

В цій програмі ввід і вивід реалізований вбудованими функціями Python. Тому нам не доведеться користуватися перенаправлення файлового потоку.

Запускаємо програму і вводимо назву файлу:

```
echo "generatedC" | python3 example_3.py
```

Як бачимо, згенерувалося два файли програмою:

```
ls | grep "^generatedC*"
```

```
generatedC.c
generatedC.h
```

Вміст файлу-заголовку:

```c
/*
 * generatedC.h
 *
 * This grammar is generated using the Grammar.export_c() method and
 * should be used with the libeler module.
 *
 * Source class: MyGrammar
 * Created at: 2020-03-31 16:18:51
 */
#ifndef CLERI_EXPORT_GENERATEDC_H_
#define CLERI_EXPORT_GENERATEDC_H_

#include <cleri/cleri.h>

cleri_grammar_t * compile_generatedC(void);

enum cleri_grammar_ids {
    CLERI_NONE,   // used for objects with no name
    CLERI_GID_K_HI,
    CLERI_GID_R_NAME,
    CLERI_GID_START,
    CLERI_END // can be used to get the enum length
};

#endif /* CLERI_EXPORT_GENERATED_H_ */
```

Вміст файлу-заголовку:

```c
/*
 * generatedC.c
 *
 * This grammar is generated using the Grammar.export_c() method and
 * should be used with the libcleri module.
 *
 * Source class: MyGrammar
 * Created at: 2020-03-31 16:18:51
 */

#include "generatedC.h"
#include <stdio.h>

#define CLERI_CASE_SENSITIVE 0
#define CLERI_CASE_INSENSITIVE 1

#define CLERI_FIRST_MATCH 0
#define CLERI_MOST_GREEDY 1

cleri_grammar_t * compile_generatedC(void)
{
    cleri_t * r_name = cleri_regex(CLERI_GID_R_NAME, "^(?:\"(?:[^\"]*)\")+");
    cleri_t * k_hi = cleri_keyword(CLERI_GID_K_HI, "hi", CLERI_CASE_SENSITIVE);
    cleri_t * START = cleri_sequence(
        CLERI_GID_START,
        2,
        k_hi,
        r_name
    );

    cleri_grammar_t * grammar = cleri_grammar(START, "^\\w+");

    return grammar;
}
```

4. ОБ’ЄКТ RESULT

Як ми раніше зазначали, об’єкт *Result* повертається з функції *parse*. Метод *as_str(translate=None)* повертає стрічку що відображає результат парсингу. Параметр *translate=None*, повинен бути функцією що приймає елемент як аргумент. Ця функція може бути використана для повернення конкретної стрічки для відповідного елементу. Якщо ж ця функція повертає порожню стрічку, то буде згенероване звичне повідомлення. Якщо ж функція повертає *None*, то стрічка не буде згенерована.

Приклад функції *translate* наведений нижче.

```
#  Функція не надасть додаткового повідомлення
def translate1(elem):
    return ''  # 'error at position x'


# Text may be returned based on gid
def translate2(elem):
    if elem is some_elem:
        return 'A'  # error at position x, expecting: A
    elif elem is other_elem:
        return ''  # error at position x
    else:
        return None  # повідомлення не згенеровано


# Приклад використання функції translate
print(my_grammar.parse('some string').as_str(translate=translate2))
```

Функція *is_valid* повертає булеве значення, що сигналізує чи стрічка є правильна згідно граматики.

Приклад використання функції *is_valid* наведений нижче.

```
res = my_grammar.parse('bye "Iris"')
print(res.is_valid)  # => False
```

Атрибут *pos* повертає позицію, де парсер повинен був зупинитись. Якщо ж *is_valid* є істинним повертає довжину стрічки, що було передано для парсингу.

Приклад використання атрибуту *pos* наведений нижче.

```
result = my_grammar.parse('hi Iris')
print(res.is_valid, result.pos)  # => False, 3
```

Атрибут ***tree*** містить спарсене дерево. Навіть якщо *is_valid* є хибою, то дерево повертається, до місця де парсинг зупинився. Дерево є вузлом, що може містити кілька дочірніх вузлів.

Приклад, що відображає структуру дерева у вигляді JSON форматі наведений нижче.

```
import json
from pyleri import Choice
from pyleri import Grammar
from pyleri import Keyword
from pyleri import Regex
from pyleri import Repeat
from pyleri import Sequence


class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    k_bye = Keyword('bye')
    START = Repeat(Sequence(Choice(k_hi, k_bye), r_name))
```

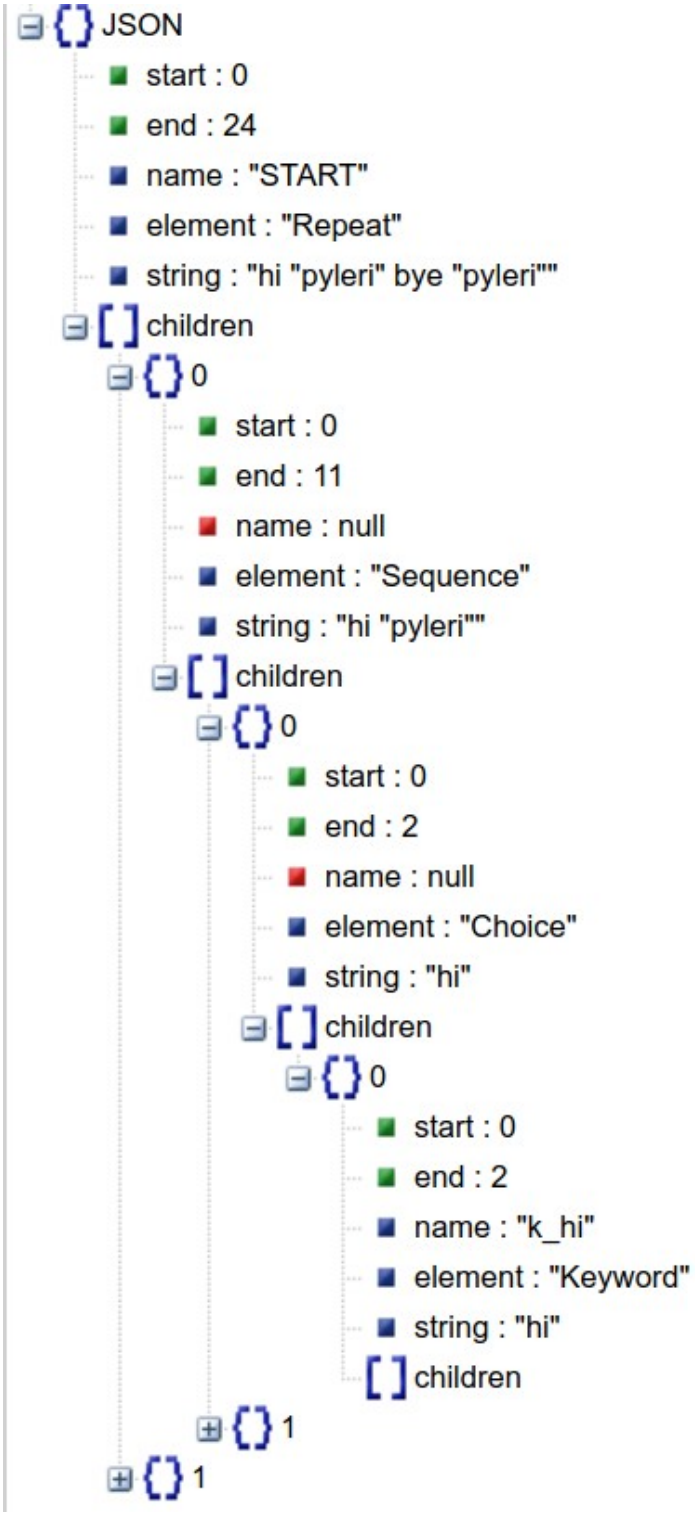
```
# Функція, що повертає вузол у вигляді словника
def node_props(node, children):
    return {
        'start': node.start,
        'end': node.end,
        'name': node.element.name if hasattr(node.element, 'name') else None,
        'element': node.element.__class__.__name__,
        'string': node.string,
        'children': children
    }


# Рекурсивна функція для отримання всіх дочірніх вузлів
def get_children(children):
    return [node_props(c, get_children(c.children)) for c in children]


# Функція для представлення дерева
def view_parse_tree(res):
    if res.tree.children:
        start = res.tree.children[0]
    else:
        start = res.tree
    return node_props(start, get_children(start.children))


if __name__ == '__main__':
    my_grammar = MyGrammar()
    res = my_grammar.parse('hi "foo" bye "foo"')
    # Спарсене дерево буде відображатись у вигляді JSON об'єкту
    print(json.dumps(view_parse_tree(res), indent=2))
```

Цього разу граматика очікує відформатовану стрічку, що може бути повторена нескінченну кількість разів. Відформатована стрічка повинна містити одне на вибір з ключових слів - *hi* або *bye* і стрічку довільної довжини.

Класи, що необхідні для реалізації цієї граматики будуть розглянуті пізніше.

Додатково у цій програмі було імпортовано вбудований модуль *json*, для роботи з JSON-об’єктами.

Тож запустивши програму, ми отримуємо результат що можна зобразити наступним чином.



Як ми можемо зауважити, кожен вузол містить п’ять атрибутів:

1. **start** : початок об’єкту вузла
2. **end** : кінець об’єкту вузла
3. **element** : об’єкт типу Element, що буде розглянуто пізніше
4. **string** : стрічка, що була передана
5. **children** : масив дочірніх вузлів

Грунтуючись на факті того що значення атрибуту *start* дочірнього вузла повинно бути більшим ніж значення атрибуту батьківського вузла і значення атрибуту дочірнього вузла *end* повинно бути меншим ніж значення атрибуту батьківського вузла, можна легше зрозуміти поняття дочірнього вузла.

Атрибут *expecting* повертає множину елементів що парсер очікує отримати в *pos*. Попри те, що *is_valid* є істинним, парсер може очікувати необов’язкові елементи. Це може бути надзвичайно корисним для реалізації генерації коду, підказування і вистежиння помилок.

Нижче наведений приклад, що показує як з допомогою цього атрибуту можна реалізувати автоматичне виправлення помилок.

```
import re
import random
from pyleri import Choice
from pyleri import Grammar
from pyleri import Keyword
from pyleri import Repeat
from pyleri import Sequence
from pyleri import end_of_statement


class MyGrammar(Grammar):
    RE_KEYWORDS = re.compile(r'\S+')
    r_name = Keyword('"pyleri"')
    k_hi = Keyword('hi')
    k_bye = Keyword('bye')
    START = Repeat(Sequence(Choice(k_hi, k_bye), r_name), mi=2)
```

```
# Друк очікуваних парсером елементів у вигляді нумерованого списку
def print_expecting(node_expecting, string_expecting):
    for loop, e in enumerate(node_expecting):
        string_expecting = '{}\n\t({}) {}'.format(string_expecting, loop, e)
    print(string_expecting)


# Перетворення стрічки, щоб вона відповідала грамматиці
def auto_correction(string, my_grammar):
    node = my_grammar.parse(string)
    print('\nСпарсена стрічка: {}'.format(node.tree.string))

    if node.is_valid:
        string_expecting = 'Стрічка є правильною. \nОчікується: '
        print_expecting(node.expecting, string_expecting)

    else:
        string_expecting = 'Стрічка є невірною.\nОчікується: ' \
            if not node.pos \
            else 'Стрічка є невірною. \nПісля "{}" очікується: '.format(
            node.tree.string[:node.pos])
        print_expecting(node.expecting, string_expecting)

        # Обираєм випадковий елемент з очікуваних парсером і
        # поєднуємо з нашою стрічкою до виявленої помилки
        selected = random.choice(list(node.expecting))
        string = '{} {}'.format(node.tree.string[:node.pos],
                                selected
                                if selected
                                   is not end_of_statement else '')

        auto_correction(string, my_grammar)


if __name__ == '__main__':
    my_grammar = MyGrammar()
    string = 'hello "pyleri"'
    auto_correction(string, my_grammar)
```

Запустивши цю програму, ми отримуємо наведений нижче результат.

```
Спарсена стрічка:  hi "pyleri"
Стрічка є невірною.
Після " hi "pyleri"" очікується:
    (0) hi
    (1) bye
на перетворення
 hi "pyleri" bye

Спарсена стрічка:  hi "pyleri" bye
Стрічка є невірною.
Після " hi "pyleri" bye" очікується:
    (0) "pyleri"
на перетворення
 hi "pyleri" bye "pyleri"

Спарсена стрічка:  hi "pyleri" bye "pyleri"
Стрічка є правильною.
Очікується:
    (0) hi
    (1) bye
```

Тож, як бачимо рекурсивна функція *auto_correction* на основі атрибуту expected генерує нову стрічку і викликає себе ж з генерованою стрічкою. Базовим випадком рекурсії є випадок, при якому стрічка є правильною з точки зору граматики.

Потрібно зауважити, що в цьому прикладі у елемент *Repeat* було додатково передано мінімальну кількість повторень, тому функція викликається три рази.

5. ВБУДОВАНІ ЕЛЕМЕНТИ МОДУЛЮ

Бібліотека підтримує кілька елементів, що всі являються дочірніми класами відносно класу *Element*, що є очевидно абстрактним.

5.1. ЕЛЕМЕНТ KEYWORD

Для співставлення з елементом *Keyword(keyword, ign_case=False)*, парсер повинен співставити всі символи, що містяться в параметрі *keyword* з урахуванням регістру чи без, що визначається другим параметром *ign_case*.

Інколи потрібно додатково вказати які символи взагалі дозволені для парсингу під час співставлення з ключовими словами. Це задається безпосередньо в класі наслідникy від *Grammar*, переозначуючи поле *RE_KEYWORDS*, що за замовчуванням використовує регулярний вираз *^[A-Za-z0-9_]+* . Що як бачимо, дозволяє тільки букви з латинського алфавіту і цифри.

Нижче наведений приклад використання цього елементу і також показано перевизначення поля *RE_KEYWORDS*.

```
import re
from pyleri import Grammar
from pyleri import Keyword


class TicTacToe(Grammar):
    # Переозначення регулярного виразу для ключових слів
    # Дозволено символи латинського алфавіту і тире
    RE_KEYWORDS = re.compile('^[A-Za-z-]+')

    START = Keyword('tic-tac-toe', ign_case=True)


ttt_grammar = TicTacToe()
print(ttt_grammar.parse('Tic-Tac-Toe').is_valid)  # => True
```

5.2. ЕЛЕМЕНТ REGEX

Елемент Regex(pattern), використовуючи вбудований модуль *re* , співставляє стрічку відповідно до шаблону, що задається першим аргументом.

5.3. ЕЛЕМЕНТ TOKEN

Токен може бути одним або більше символів, що зазвичай використовується для співставлення операторів, таких як: +, -, // тощо.

Приклад використання цього елементу наведено нижче.

```
from pyleri import Grammar
from pyleri import Keyword
from pyleri import Token
from pyleri import List


class Ni(Grammar):
    t_dash = Token('-')
    START = List(Keyword('ni'), delimiter=t_dash)


ni = Ni()
print(ni.parse('ni-ni-ni-ni-ni').is_valid)  # => True
```

За замовчуванням всі стрічки автоматично конвертуються в токени, тому можна було без явного створення об’єкту типу *Token* передати стрічку як параметр *delimiter*.

5.4. ЕЛЕМЕНТ TOKENS

Tokens(tokens) може бути використаний для реєстрації кілька токенів одночасно. Аргумент *token* є стрічкою, що містить токени розділені пробілами. Якщо ці токени мають різну довжину, то парсер намагатиметься співставити найдовший з них.

Приклад використання цього елементу наведений нижче.

```
from pyleri import Grammar
from pyleri import Keyword
from pyleri import Tokens
from pyleri import List


class Ni(Grammar):
    tks = Tokens('+ - !=')
    START = List(Keyword('ni'), delimiter=tks)


ni = Ni()
print(ni.parse('ni + ni != ni - ni').is_valid)  # => True
```

5.5. ЕЛЕМЕНТ SEQUENCE

Елемент *Sequence(element, element, ...)* співставляє кожен елемент, що передається у функцію. Кількість переданих елементів є довільною.

Приклад використання цього елементу наведений нижче.

```
from pyleri import Grammar
from pyleri import Keyword
from pyleri import Sequence


class TicTacToe(Grammar):
    START = Sequence(Keyword('Tic'), Keyword('Tac'), Keyword('Toe'))


ttt_grammar = TicTacToe()
print(ttt_grammar.parse('Tic Tac Toe').is_valid)  # => True
```

5.6. ЕЛЕМЕНТ CHOICE

Choice(element, element, ..., most_greedy=True) обирає один з переданих у функцію елементів. Механізм роботи дещо різниться відносно переданого необов'язково параметру *most_greedy*.

Якщо *most_greedy* є хибним, то обирається перший елемент, що співпав. В іншому випадку найдовше співпавший елемент. Тож можна дещо збільшити швидкодію вказавши його хибним.

Приклад використання цього елементу наведений нижче.

```
from pyleri import Grammar
from pyleri import Regex
from pyleri import Keyword
from pyleri import Sequence
from pyleri import Choice


class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    k_bye = Keyword('bye')
    START = Sequence(Choice(k_hi, k_bye), r_name)


my_grammar = MyGrammar()
print(my_grammar.parse('hi "Iris"').is_valid)  # => True
print(my_grammar.parse('bye "Iris"').is_valid)  # => True
```

5.7. ЕЛЕМЕНТ REPEAT

Repeat(element, mi=0, ma=None) потребує повторення елементу, що задається першим аргументом мінімум *mi* разів та максимум *ma* разів. Якщо *ma* встановлений значенням *None*, то елемент може повторюватись нескінченно. Також існує очевидне обмеження на те, що *mi* не може бути більшим *ma*.

Приклад використання цього елементу наведений нижче.

```
from pyleri import Repeat, Keyword, Grammar


class Ni(Grammar):
    START = Repeat(Keyword('ni'))


ni = Ni()
print(ni.parse('ni ni ni ni ni').is_valid)  # => True
```

5.8. ЕЛЕМЕНТ LIST

List(element, delimiter=',' , mi=0, ma=None, opt=False) є аналогом *Repeat*. Однією з відмінностей - наявністю розділювача *delimiter*. Також за допомогою параметра *opt* можна вказати чи може стрічка закінчуватись на розділювач (за замовчуванням не може).

Приклад використання цього елементу наведений нижче.

```
from pyleri import List,Keyword,Grammar


class Ni(Grammar):
    START = List(Keyword('ni'))


ni = Ni()
print(ni.parse('ni, ni, ni, ni, ni').is_valid)  # => True
```

5.9. ЕЛЕМЕНТ OPTIONAL

Optional(element) є необов’язковим при парсингу елементом і є по суті більш читабельним і швидшим синонімом до *Repeat(element, 0, 1)*.

Приклад використання цього елементу наведений нижче.

```
from pyleri import Grammar,Keyword,Regex,Sequence,Optional


class MyGrammar(Grammar):
    r_name = Regex('(?:"(?:[^"]*)")+')
    k_hi = Keyword('hi')
    START = Sequence(k_hi, Optional(r_name))


my_grammar = MyGrammar()
print(my_grammar.parse('hi "Iris"').is_valid)  # => True
print(my_grammar.parse('hi').is_valid)  # => True
```

5.10. ЕЛЕМЕНТ REF

Ref() надає можливість створювати посилання на поточні елементи для реалізації рекурсії.

Приклад реалізації рекурсивного списку наведено нижче.

```
from pyleri import Grammar, Ref, Choice, Keyword, Sequence, List


class NestedNi(Grammar):
    START = Ref()
    ni_item = Choice(Keyword('ni'), START)
    START = Sequence('[', List(ni_item), ']')


nested_ni = NestedNi()
print(nested_ni.parse('[ni, ni, [ni, [], [ni, ni]]]').is_valid)  # => True
```

5.11. ЕЛЕМЕНТ PRIO

Prio(element, element, ...) обирає перший з елементів, що співпадає і дозволяє *THIS* виконувати рекурсивні операції. З допомогою *THIS* ми вказуємо на елемент типу *Prio*.

Приклад використання цього елементу наведено нижче.

```
from pyleri import Grammar, Keyword, Prio, Sequence, THIS


class Ni(Grammar):
    k_ni = Keyword('ni')
    START = Prio(
        k_ni,
        Sequence('(', THIS, ')'),
        Sequence(THIS, Keyword('or'), THIS),
        Sequence(THIS, Keyword('and'), THIS))


ni = Ni()
print(ni.parse('(ni or ni) and (ni or ni)').is_valid)  # => True
```

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pyleri - Python Parser [Електронний ресурс]. - Режим доступу: https://github.com/transceptor-technology/pyleri
2. Python Pip [Електронний ресурс]. - Режим доступу: https://en.wikipedia.org/wiki/Pip_(package_manager)
3. Lexical analysis [Електронний ресурс]. - Режим доступу: https://en.wikipedia.org/wiki/Lexical_analysis
4. Keith D. Cooper and Linda Torczon. Engineering a Compiler. 2012 Elsevier, Inc., 2012. –800p
5. Andrew W. Appel, Maia Ginsburg. Modern Compiler implementation in C. Cambridge University press, 1998. –544 c.
6. Allen I. Holub. Compiler design in C. Prentice-Hall, Inc. , 1990. –984 p.
7. Jean-Paul Tremblay, Paul G. Sorenson. The theory and practice of compiler writing. McGraw-Hill Book Company, 1985. –796 p.
8. Ахо Альфред, Лам Моника С., Сети Рави, Ульман Джеффри. Компиляторы: принципы, технологии, инструментарий. Пер. с англ. –М.: Изд. дом “Вильямс”, 2008. –1184 с.