МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Прикарпатський національний університет імені Василя Стефаника

Реферат з дисципліни “ Системне програмування ”

на тему: PY PEG.

Виконав:

Кирилюк В.Б.

КІ-31

Івано-Франківськ 2020р

ВСТУП І УСТАНОВКА

Python - це дуже зручна сценарна мова. Вона навіть надає вам доступ до власного парсера та компілятора. Також надає вам доступ до різних інших аналізаторів для спеціальних цілей, таких як XML та рядкові шаблони.

Але іноді ви можете визначити власний парсер. Це те, для чого pyPEG. І pyPEG підтримує Unicode.

pyPEG - це звичайний та простий парсер синтаксичного аналізатора для версій Python 2.7 та 3.x. Він заснований на граматиці виразного розбору, PEG. За допомогою pyPEG ви можете дуже просто розбирати багато формальних мов. Як це працює?

Установка

Ви можете встановити випуск pyPEG серії 2.x з PyPY за допомогою:

```
pip install pypeg2
```

PEG - це щось на зразок регулярних виразів з рекурсією. Граматики схожі на шаблони. Давайте зробимо приклад. Скажімо, ви хочете проаналізувати декларацію функції на мові, подібній C. Така декларація функції складається з:

- type declaration
- name
- parameters
- block with instructions

декларація типу , ім'я , параметри , блок з інструкціями.
За допомогою pyPEG ви оголошуєте клас Python для кожного типу об'єкта, який ви хочете проаналізувати. Потім цей клас інстанціюється для кожного розібраного об'єкта. Цей клас отримує граматику атрибутів з описом того, що слід проаналізувати яким чином. У нашому простому прикладі ми підтримуємо дві різні речі, які в нашій мові оголошуються як ключові слова: int та long.

Отже, ми пишемо декларацію класу для набору тексту, яка підтримує Enum з двох можливих ключових слів як своєї граматики:

```
class Type(Keyword):
    grammar = Enum( K("int"), K("long") )
```

Загальні завдання розбору включені в рамку pyPEG. У цьому прикладі ми використовуємо клас ключових слів, оскільки результатом буде ключове слово, а ми використовуємо об’єкти ключових слів (із абревіатурою K), оскільки те, що ми аналізуємо, буде одним із ключових слів, що зараховані до списку.

Загальний результат буде функцією. Отже ми оголошуємо клас функцій:

```
class Function:
    grammar = Type, …
```

Наступним буде ім'я функції Функція розбору. Імена дещо особливі в pyPEG. Але з ними легко впоратися: для розбору імені є готова функція name (), яку можна зателефонувати у вашій граматиці для створення .name Attribute:

```
class Function:
    grammar = Type, name(), …
```

Тепер для частини Параметри. Спочатку оголосимо клас для параметрів. Параметри повинні бути колекцією, оскільки їх може бути багато. pyPEG має кілька готових колекцій. Для випадку параметрів підійде колекція простору імен. Він забезпечує індексований доступ за іменем, а Параметри мають імена (у нашому прикладі: a і b). Ми пишемо це так:

```
class Parameters(Namespace):
    grammar = …
```

Єдиний Параметр має саму структуру. Він має Тип і ім'я (). Тож давайте визначимось:

```
class Parameter:
    grammar = Type, name()

class Parameters(Namespace):
    grammar = …
```

pyPEG інстанціює клас Параметри для кожного розібраного параметра. Куди піде Тип? Функція name () буде генерувати атрибут .name, але об'єкт Type? Що

ж, давайте перемістимо його до атрибута, також названого .typing. Щоб створити атрибут, pyPEG пропонує функцію attr ():

```
class Parameter:
    grammar = attr("typing", Type), name()

class Parameters(Namespace):
    grammar = …
```

До речі: name () - це лише ярлик attr ("ім'я", Symbol). Він породжує символ.

Як ми можемо заповнити нашу колекцію простору імен з назвою Параметри? Ну, ми повинні заявити, як буде виглядати список об’єктів Параметра у нашому вихідному тексті. Простий шлях пропонує pyPEG з функціями кардинальності. У цьому випадку ми можемо використовувати Maybe_some (). Ця функція відображає зірочність кардинальності, *

```
class Parameter:
    grammar = attr("typing", Type), name()

class Parameters(Namespace):
    grammar = Parameter, maybe_some(",", Parameter)
```

Ось так ми виражаємо список, розділений комами. Оскільки це завдання настільки поширене, знову є функція генератора ярликів, csl (). Код нижче буде виконувати те саме, що і вище:

```
class Parameter:
    grammar = attr("typing", Type), name()

class Parameters(Namespace):
    grammar = csl(Parameter)
```

Можливо, функція не має параметрів. Це справа, яку ми повинні розглядати. Що тоді має статися? У нашому прикладі тоді простір імен параметрів повинен бути порожнім. Для цього випадку ми використовуємо іншу функцію кардинальності (необов’язково (). Він являє собою кардинальність питання питання,?

```
class Parameter:
    grammar = attr("typing", Type), name()

class Parameters(Namespace):
    grammar = optional(csl(Parameter))
```

Ми можемо продовжувати наш клас функцій. Параметри будуть в дужках, ми просто вкладаємо це в граматику:

```
class Function:
    grammar = Type, name(), "(", Parameters, ")", …
```

Тепер про блок інструкцій. Ми можемо оголосити інший збірник для Інструкцій. Але сама функція може розглядатися як список інструкцій. Тож давайте оголосимо це так. Спочатку ми робимо сам клас функцій списком:

```
class Function(List):
    grammar = Type, name(), "(", Parameters, ")", …
```

Якщо класом є Список, pyPEG помістить все в цей список, який буде проаналізований і не генерує атрибут. Тож із цією модифікацією наші параметри тепер теж будуть внесені до цього списку. Так буде і Тип. Це варіант, але в нашому прикладі це не те, що ми хочемо. Тож давайте перемістимо їх до атрибутів .typing та Attribute .parms відповідно:

```
class Function(List):
    grammar = attr("typing", Type), name(), \
            "(", attr("parms", Parameters), ")", …
```

Тепер ми можемо визначити, як буде виглядати блок, і поставити його просто позаду в граматику функції. Клас Інструкції у нас зрозумілий і простий. Звичайно, у прикладі реального світу це може бути досить складно ;-) Ось ми його просто маємо як слово. Слово - це попередньо визначений RegEx; це re.compile (r "\ w +").

```
class Instruction(str):
    grammar = word, ";"

block = "{", maybe_some(Instruction), "}"
```

Тепер давайте поставимо це до хвоста нашої Function.grammar:

```
class Function(List):
    grammar = attr("typing", Type), name(), \
            "(", attr("parms", Parameters), ")", block
```

Caveat:

pyPEG 2.x написано для Python 3. Ви можете використовувати його з Python 2.7 при наступному імпорті (вам це не потрібно для Python 3):

з __future__ імпортувати unicode_literals, print_function

Що ж, зараз це виглядає досить добре. Давайте спробуємо це за допомогою функції parse ():

```
>>> from pypeg2 import *
>>> class Type(Keyword):
...     grammar = Enum( K("int"), K("long") )
...
>>> class Parameter:
...     grammar = attr("typing", Type), name()
...
>>> class Parameters(Namespace):
...     grammar = optional(csl(Parameter))
...
>>> class Instruction(str):
...     grammar = word, ";"
...
>>> block = "{", maybe_some(Instruction), "}"
>>> class Function(List):
...     grammar = attr("typing", Type), name(), \
...             "(", attr("parms", Parameters), ")", block
...
>>> f = parse("int f(int a, long b) { do_this; do_that; }",
...         Function)
>>> f.name
Symbol('f')
>>> f.typing
Symbol('int')
>>> f.parms["b"].typing
Symbol('long')
>>> f[0]
'do_this'
>>> f[1]
'do_that'
```

Складання тексту

pyPEG може зробити більше. Це не тільки рамка для розбору тексту, вона може також створювати вихідний код. Граматика pyPEG - це не лише "як" шаблон,

вона може бути фактично використана як шаблон для складання тексту. Просто зателефонуйте до функції compose ():

```
>>> compose(f, autoblank=False)
'intf(inta, longb){do_this;do_that;}'
```

Як бачите, для складання спочатку бракує пробілу. Це тому, що ми використовували автоматичне видалення пробілів білого простору під час розбору (який увімкнено за замовчуванням), але ми відключили автоматичне додавання пробілів, якщо порушуємо синтаксис інакше. Щоб покращити це, ми повинні трохи розширити наші шаблони граматики. У цьому випадку в pyPEG є об'єкти функції зворотного виклику. Вони виконуються лише через compose () та ігноруються парсером (). І як завжди, для загальних випадків є наперед визначені. Давайте спробуємо це. Спочатку додамо пробіл між речами, які слід розділити:

```
class Parameter:
    grammar = attr("typing", Type), blank, name()

class Function(List):
    grammar = attr("typing", Type), blank, name(), \
            "(", attr("parms", Parameters), ")", block
```

Після скидання все це призведе до виводу:

```
>>> compose(f, autoblank=False)
'int f(int a, long b){do_this;do_that;}'
```

The `blank` after the comma `int a, long b` was generated by the `csl()` function; `csl(Parameter)` generates:

```
Parameter, maybe_some(",", blank, Parameter)
```

Текст відступу

На мовах, подібних C (як у нашому прикладі), ми любимо вводити блоки відступу. Відступ - це щось, що є відносно поточного положення. Якщо щось вже є в блоці і має бути відступним, його слід відступити два рази (і так далі). У цьому випадку pyPEG має систему з відступом.

Система відступу в основному використовує відтворення функції () та об'єкт функції зворотного виклику endl. З відступом ми можемо відзначити, що має

бути відступним, надсилаючи endl означає, що тут слід починати наступний рядок вихідного коду. Ми можемо використовувати це для нашого блоку:

```
class Instruction(str):
    grammar = word, ";", endl

block = "{", endl, maybe_some(indent(Instruction)), "}", endl

class Function(List):
    grammar = attr("typing", Type), blank, name(), \
            "(", attr("parms", Parameters), ")", endl, block
```

Це змінює вихід на:

```
>>> print(compose(f))
int f(int a, long b)
{
    do_this;
    do_that;
}
```

Функції зворотного виклику, визначені користувачем

Завдяки визначеним користувачем функціям зворотного виклику pyPEG пропонує необхідну гнучкість, щоб бути корисною як шаблонна система загального призначення для генерації коду. У нашому простому прикладі скажімо, що ми хочемо, щоб інформація про обробку в коментарях у Декларації про функцію, тобто рівень відступу в коментарі, була кожна інструкція. Для цього ми можемо визначити власну функцію зворотного виклику:

```
class Instruction(str):
    def heading(self, parser):
        return "/* on level " + str(parser.indention_level) \
                + " */", endl
```

Така функція зворотного виклику викликається двома аргументами. Перший аргумент - об'єкт для виведення. Другий аргумент - це об'єкт парсера для отримання інформації про стан процесу складання. Оскільки це відповідає умові для методів Python, ви можете записати його як метод класу, до якого він належить.

Повернене значення такої функції зворотного виклику повинно бути текстом, що виникає. У нашому прикладі генерується оболонка C коментарів із примітками. Ми можемо це зараз вкласти в граматику.

```
class Instruction(str):
    def heading(self, parser):
        return "/* on level " + str(parser.indention_level) \
                + " */", endl

    grammar = heading, word, ";", endl
```

Результат відповідний:

```
>>> print(compose(f))
int f(int a, long b)
{
    /* on level 1 */
    do_this;
    /* on level 1 */
    do_that;
}
```

XML output

Іноді ви хочете обробити те, що ви розібрали з ланцюжком інструментів XML або з ланцюжком інструментів YML. Через це в pyPEG є резервний файл XML. Просто викличте функцію thing2xml (), щоб отримати байти із закодованим XML:

```
>>> from pypeg2.xmlast import thing2xml
>>> print(thing2xml(f, pretty=True).decode())
<Function typing="int" name="f">
  <Parameters>
    <Parameter typing="int" name="a"/>
    <Parameter typing="long" name="b"/>
  </Parameters>
  <Instruction>do_this</Instruction>
  <Instruction>do_that</Instruction>
</Function>
```