Міністерство освіти і науки України
Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Фізико-технічний факультет
Кафедра комп’ютерної інженерії та електроніки

# Реферат

З дисципліни „Системне програмування”
на тему
“Генератор парсерів CANOPY”

Виконав:
студент групи КІ-31
Доцяк В. І.

м. Івано-Франківськ – 2020 рік

**ЗМІСТ**

# ВСТУП

Canopy – це генератор парсерів з граматикою PEG (Parsing expression grammar), який дозволяє компілювати файл, що описує граматику виразів синтаксичного аналізу в модуль розбору на цільовій мові використовуючи простий короткий синтаксис. Canopy підтримує мови програмування Java, JavaScript, Python та Ruby і використовує алгоритм парсингу Packrat.

Граматика Canopy має чітку особливість використання анотацій про дії для використання спеціального коду в аналізаторі. На практиці ви просто записуєте назву функції поруч із правилом, а потім реалізуєте функцію у вихідному коді.

Програма Canopy - це вільне програмне забезпечення, тобто її можна розповсюджувати чи редагувати відповідно до умов загальної публічної ліцензії GNU, опублікованої Фондом вільного програмного забезпечення.

Щоб встановити Canopy, потрібно у консолі набрати рядок:

```
$ npm install -g canopy
```

РОЗДІЛ 1

# Використання генератора парсерів Canopy з різними мовами програмування

Canopy підтримує мови програмування Java, JavaScript, Python та Ruby.

## 1.1. Використання Canopy з Java

Щоб краще зрозуміти використання Canopy з Java далі будуть наведені приклади спрощеної граматики для URL.

### `url.peg`

```
grammar URL

url       <-  scheme "://" host pathname search hash?
scheme    <-  "http" "s"?
host      <-  hostname port?
hostname  <-  segment ("." segment)*
segment   <-  [a-z0-9-]+
port      <-  ":" [0-9]+
pathname  <-  "/" [^ ?]*
search    <-  ("?" query:[^ #]*)?
hash      <-  "#" [^ ]*
```

Цю граматику можна компілювати в пакет Java використовуючи `canopy`:

```
$ canopy url.peg --lang java
```

Це створить пакет під назвою `url`, містить всю логіку аналізатора. Назва пакету заснована на шляху до `.peg` файлу. Для прикладу, якщо запустити `canopy`:

```
$ canopy com/jcoglan/canopy/url.peg --lang java
```

то ви отримаєте пакет під назвою `com.jcoglan.canopy.url`.

```
import url.URL;
import url.TreeNode;
import url.ParseError;

public class Example {
    public static void main(String[] args) throws ParseError {
        TreeNode tree = URL.parse("http://example.com/search?
q=hello#page=1");

        for (TreeNode node : tree.elements) {
            System.out.println(node.offset + ", " + node.text);
        }

        /*  prints:

            0, http
            4, ://
            7, example.com
            18, /search
            25, ?q=hello
            33, #page=1       */
    }
}
```

Цей приклад показує декілька важливих речей:

Можна викликати аналізатор, використовуючи функцію `parse()`

Метод `parse()`повертає дерево вузлів.

Кожен вузол має три властивості:

- `String text` (текстовий рядок) – фрагмент вхідного тексту, який відображає вузол
- `Int offset` – кількість символів у вхідному тексті, при яких з'являється вузол
- `List<TreeNode> elements` – масив вузлів, які відповідають підвиразам

Можна використовувати вираз `elements` для входу в структуру дерева, чи використати створені Canopy мітки, які можуть зробити код більш зрозумілим.

```
import url.URL;
import url.TreeNode;
import url.ParseError;
import url.Label;

public class Example {
    public static void main(String[] args) throws ParseError {
        TreeNode tree = URL.parse("http://example.com/search?
q=hello#page=1");

System.out.println(tree.elements.get(4).elements.get(1).text);
        // -> 'q=hello'

System.out.println(tree.get(Label.search).get(Label.query).text);
        // -> 'q=hello'
    }
}
```

## Помилки аналізатора

Якщо надати аналізатору вхідний текст, який не відповідає граматиці, `ParseError` видасть:

```
import url.URL;
import url.TreeNode;
import url.ParseError;

public class Example {
    public static void main(String[] args) throws ParseError {
        TreeNode tree = URL.parse("https://example.com./");
    }
}

/*  prints:
```

```
    url.ParseError: Line 1: expected [[a-z0-9-]]
    https://example.com./
                        ^                          */
```

**Реалізація дій**

Допустимо, у вас є граматика, яка використовує примітки про дії, наприклад:

maps.peg

```
grammar Maps

  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

У Java при компілюванні вищезазначеної граматики створюється пакет під назвою `maps`, який містить класи під назвою `Maps`, `TreeNode` та `ParseError`, перерахунок під назвою `Label` та інтерфейс під назвою `Actions`. Ви надаєте функції дії в аналізатор, реалізуючи інтерфейс `Action`, який має один метод для кожної дії, названої в граматиці, кожна з яких повинна повернути `TreeNode`. `TreeNode` має конструктор без аргументів, тому зробити підкласи з нього відносно просто.

Наступний приклад аналізує вхідні дані `{'ints': [1,2,3]}`. Він визначає один підклас `TreeNode` для кожного типу значень у дереві:

- Pair (пари) `wraps a` Map<String, List<Integer>>
- Text (текст) `wraps a` String
- Array (масив) `wraps a` List<Integer>
- Number (число) `wraps an` int

Потім він реалізує інтерфейс `Action` для генерації значень цих типів з аналогів аналізатора.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import maps.Actions;
import maps.Label;
import maps.Maps;
import maps.ParseError;
import maps.TreeNode;

class Pair extends TreeNode {
    Map<String, List<Integer>> pair;

    Pair(String key, List<Integer> value) {
        pair = new HashMap<String, List<Integer>>();
        pair.put(key, value);
    }
}

class Text extends TreeNode {
    String string;

    Text(String string) {
        this.string = string;
    }
}

class Array extends TreeNode {
    List<Integer> list;

    Array(List<Integer> list) {
        this.list = list;
```

```
    }
}

class Number extends TreeNode {
    int number;

    Number(int number) {
        this.number = number;
    }
}

class MapsActions implements Actions {
    public Pair make_map(String input, int start, int end,
List<TreeNode> elements) {
        Text string = (Text)elements.get(1);
        Array array = (Array)elements.get(3);
        return new Pair(string.string, array.list);
    }

    public Text make_string(String input, int start, int end,
List<TreeNode> elements) {
        return new Text(elements.get(1).text);
    }

    public Array make_list(String input, int start, int end,
List<TreeNode> elements) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(((Number)elements.get(1)).number);
        for (TreeNode el : elements.get(2)) {
            Number number = (Number)el.get(Label.value);
            list.add(number.number);
        }
        return new Array(list);
    }

    public Number make_number(String input, int start, int end,
List<TreeNode> elements) {
```

```
        return new Number(Integer.parseInt(input.substring(start,
end), 10));
    }
}

public class Example {
    public static void main(String[] args) throws ParseError {
        Pair result = (Pair)Maps.parse("{'ints':[1,2,3]}", new
MapsActions());

        System.out.println(result.pair);
        // -> {ints=[1, 2, 3]}
    }
}
```

Використання анотації граматики <Type> не підтримується у версії Java.

1.2. Використання Canopy з JavaScript

Щоб краще зрозуміти використання Canopy з JavaScript далі будуть наведені приклади спрощеної граматики для URL.

url.peg

```
grammar URL

url       <-  scheme "://" host pathname search hash?
scheme    <-  "http" "s"?
host      <-  hostname port?
hostname  <-  segment ("." segment)*
segment   <-  [a-z0-9-]+
port      <-  ":" [0-9]+
pathname  <-  "/" [^ ?]*
search    <-  ("?" query:[^ #]*)?
hash      <-  "#" [^ ]*
```

Цю граматику можна компілювати в модуль JavaScript використовуючи `canopy`:

```
$ canopy url.peg --lang js
```

Це створить файл під назвою `url.js`, який містить всю логіку аналізатора і може працювати у вузлі, та у браузері. Приклад:

```
var url = require('./url');

var tree = url.parse('http://example.com/search?q=hello#page=1');

tree.elements.forEach(function(node) {
  console.log(node.offset, node.text);
});

/*  prints:

    0 'http'
    4 '://'
    7 'example.com'
    18 '/search'
    25 '?q=hello'
    33 '#page=1'      */
```

Цей приклад показує декілька важливих речей:

Можна викликати аналізатор, використовуючи модуль `parse()`

В браузері можна викликати `URL.parse()`до використання `require()`; Canopy створює глобальну назву після граматики.

Кожен вузол має три властивості:

- `text` (текстовий рядок) – фрагмент вхідного тексту, який відображає вузол
- `offset` – кількість символів у вхідному тексті, при яких з’являється вузол

- `elements` – масив вузлів, які відповідають підвиразам

Можна використовувати вираз `elements` для входу в структуру дерева:

```
console.log(tree.elements[4].elements[1].text);
// -> 'q=hello'
```

Також, можна використати створені Canopy мітки, які можуть зробити код більш зрозумілим:

```
console.log(tree.search.query.text);
// -> 'q=hello'
```

### Помилки аналізатора

Якщо надати аналізатору вхідний текст, який не відповідає граматиці, `ParseError` видасть:

```
url.parse('https://example.com./');

/* SyntaxError: Line 1: expected [a-z0-9-]
   https://example.com./
                        ^                    */
```

### Реалізація дій

Допустимо, у вас є граматика, яка використовує примітки про дії, наприклад:

`maps.peg`

```
grammar Maps
  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
```

```
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

У JavaScript ви надаєте функції дії парсеру, використовуючи параметр `action`, який повинен бути об'єктом, що реалізує названі дії:

```
var maps = require('./maps');

var actions = {
  make_map: function(input, start, end, elements) {
    var map = {};
    map[elements[1]] = elements[3];
    return map;
  },

  make_string: function(input, start, end, elements) {
    return elements[1].text;
  },

  make_list: function(input, start, end, elements) {
    var list = [elements[1]];
    elements[2].forEach(function(el) { list.push(el.value) });
    return list;
  },

  make_number: function(input, start, end, elements) {
    return parseInt(input.substring(start, end), 10);
  }
};

var result = maps.parse("{'ints':[1,2,3]}", {actions: actions});

console.log(result);
// -> { ints: [ 1, 2, 3 ] }
```

## Розширені типи вузлів

Допустимо, у вас є граматика, яка містить примітки про типи:

words.peg

```
grammar Words
  root  <-  first:"foo" second:"bar" <Extension>
```

Щоб мати змогу використати цей синтаксичний аналіз, ви повинні передати об'єкт, що містить реалізацію названих типів через параметр `type`. Кожен визначений тип містить методи, які будуть додані до вузлів.

```
var words = require('./words');

var types = {
  Extension: {
    convert: function() {
      return this.first.text + this.second.text.toUpperCase();
    }
  }
};

words.parse('foobar', {types: types}).convert()
    // -> 'fooBAR'
```

1.3. Використання Canopy з Python

Щоб краще зрозуміти використання Canopy з Python далі будуть наведені приклади спрощеної граматики для URL:

url.peg

```
grammar URL

url       <-  scheme "://" host pathname search hash?
scheme    <-  "http" "s"?
host      <-  hostname port?
```

```
hostname  <-  segment ("." segment)*
segment   <-  [a-z0-9-]+
port      <-  ":" [0-9]+
pathname  <-  "/" [^ ?]*
search    <-  ("?" query:[^ #]*)?
hash      <-  "#" [^ ]*
```

Цю граматику можна компілювати в модуль Python використовуючи `canopy`:

```
$ canopy url.peg --lang python
```

Це створить файл під назвою `url.py`, який містить всю логіку аналізатора і може працювати у вузлі, та у браузері. Приклад:

```python
import url

tree = url.parse('http://example.com/search?q=hello#page=1')

for node in tree.elements:
    print node.offset, node.text

#   prints:

#   0 http
#   4 ://
#   7 example.com
#   18 /search
#   25 ?q=hello
#   33 #page=1
```

Цей приклад показує декілька важливих речей:

Можна викликати аналізатор, використовуючи модуль `parse()`

Метод `parse()`повертає вузли.

Кожен вузол має три властивості:

- `text` (текстовий рядок) – фрагмент вхідного тексту, який відображає вузол

- `offset` – кількість символів у вхідному тексті, при яких з'являється вузол
- `elements` – масив вузлів, які відповідають підвиразам

Можна використовувати вираз `elements` для входу в структуру дерева:

```
print tree.elements[4].elements[1].text
# -> 'q=hello'
```

Також, можна використати створені Canopy мітки, які можуть зробити код більш зрозумілим:

```
print tree.search.query.text
# -> 'q=hello'
```

**Помилки аналізатора**

Якщо надати аналізатору вхідний текст, який не відповідає граматиці, `ParseError` видасть:

```
url.parse('https://example.com./')

#   url.ParseError: Line 1: expected [a-z0-9-]
#   https://example.com./
#                        ^
```

**Реалізація дій**

Допустимо, у вас є граматика, яка використовує примітки про дії, наприклад:

`maps.peg`

```
grammar Maps

  map     <-  "{" string ":" value "}" %make_map
```

```
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

У Python ви надаєте функції дії парсеру, використовуючи параметр `action`, який повинен бути об'єктом, що реалізує названі дії:

```
import maps

class Actions(object):
    def make_map(self, input, start, end, elements):
        return {elements[1]: elements[3]}

    def make_string(self, input, start, end, elements):
        return elements[1].text

    def make_list(self, input, start, end, elements):
        list = [elements[1]]
        for el in elements[2]:
            list.append(el.value)
        return list

    def make_number(self, input, start, end, elements):
        return int(input[start:end], 10)

result = maps.parse("{'ints':[1,2,3]}", actions=Actions())

print result
# -> {'ints': [1, 2, 3]}
```

### Розширені типи вузлів

Допустимо, у вас є граматика, яка містить примітки про типи:

```
words.peg
grammar Words
```

```
root  <-  first:"foo" second:"bar" <Extension>
```

Щоб мати змогу використати цей синтаксичний аналіз, ви повинні передати об'єкт, що містить реалізацію названих типів через параметр `type`. Кожен визначений тип містить методи, які будуть додані до вузлів.

Також, можна імпортувати типи з модуля:

```
# node_types.py

class Extension(object):
    def convert(self):
        return self.first.text + self.second.text.upper()

# example.py

import words
import node_types

words.parse('foobar', types=node_types).convert()
# -> 'fooBAR'
```

Або можна долучити класи розширень до іншого класу, який ви передаєте в аналізатор:

```
import words

class Types
    class Extension(object):
        def convert(self):
            return self.first.text + self.second.text.upper()

words.parse('foobar', types=Types).convert()
# -> 'fooBAR'
```

1.4. Використання Canopy з Ruby

Щоб краще зрозуміти використання Canopy з Ruby далі будуть наведені приклади спрощеної граматики для URL:

### url.peg

```
grammar URL

url       <-  scheme "://" host pathname search hash?
scheme    <-  "http" "s"?
host      <-  hostname port?
hostname  <-  segment ("." segment)*
segment   <-  [a-z0-9-]+
port      <-  ":" [0-9]+
pathname  <-  "/" [^ ?]*
search    <-  ("?" query:[^ #]*)?
hash      <-  "#" [^ ]*
```

Цю граматику можна компілювати в модуль Ruby використовуючи `canopy`:

```
$ canopy url.peg --lang ruby
```

Це створить файл під назвою `url.rb`, який містить всю логіку аналізатора. Приклад:

```ruby
require './url'

tree = URL.parse('http://example.com/search?q=hello#page=1')

tree.elements.each do |node|
  puts node.offset, node.text
end

#   prints:
```

```
#   0 http
#   4 ://
#   7 example.com
#   18 /search
#   25 ?q=hello
#   33 #page=1
```

Цей приклад показує декілька важливих речей:

Можна викликати аналізатор, використовуючи модуль `parse()`

Метод `parse()`повертає вузли.

Кожен вузол має три властивості:

- `text` (текстовий рядок) – фрагмент вхідного тексту, який відображає вузол
- `offset` – кількість символів у вхідному тексті, при яких з'являється вузол
- `elements` – масив вузлів, які відповідають підвиразам

Можна використовувати вираз `elements` для входу в структуру дерева:

```
puts tree.elements[4].elements[1].text
# -> 'q=hello'
```

Також, можна використати створені Canopy мітки, які можуть зробити код більш зрозумілим:

```
puts tree.search.query.text
# -> 'q=hello
```

**Помилки аналізатора**

Якщо надати аналізатору вхідний текст, який не відповідає граматиці, `ParseError` видасть:

```
URL.parse('https://example.com./')

#   Line 1: expected [a-z0-9-] (URL::ParseError)
#   https://example.com./
#                        ^
```

**Реалізація дій**

Допустимо, у вас є граматика, яка використовує примітки про дії, наприклад:

maps.peg

```
grammar Maps

  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

У Ruby ви надаєте функції дії аналізатору, використовуючи параметр `action`, який повинен бути об'єктом, що реалізує названі дії:

```
require './maps'

class Actions
  def make_map(input, start, _end, elements)
    {elements[1] => elements[3]}
  end

  def make_string(input, start, _end, elements)
    elements[1].text
  end

  def make_list(input, start, _end, elements)
    list = [elements[1]]
    elements[2].each { |el| list << el.value }
    list
  end

  def make_number(input, start, _end, elements)
    input[start..._end].to_i(10)
  end
end
```

```
result = Maps.parse("{'ints':[1,2,3]}", :actions => Actions.new)

p result
# -> {"ints"=>[1, 2, 3]}
```

### Розширені типи вузлів

Допустимо, у вас є граматика, яка містить примітки про типи:

words.peg

```
grammar Words
root  <-  first:"foo" second:"bar" <Extension>
```

Щоб мати змогу використати цей синтаксичний аналіз, ви повинні передати об'єкт, що містить реалізацію названих типів через параметр `type`. Кожен визначений тип містить методи, які будуть додані до вузлів.

```
require './words'

module Types
  module Extension
    def convert
      first.text + second.text.upcase
    end
  end
end

Words.parse('foobar', :types => Types).convert
# -> 'fooBAR'
```

# РОЗДІЛ 2

# Синтаксис і семантика генератора парсерів Canopy

## 2.1. Граматичний синтаксис

Визначення граматики Canopy записуються за допомогою стандартних позначень PEG та зберігаються у файлах із розширенням `.peg`. Вони вказують тільки статичну граматику мови і не містять вбудованого коду обробки. Однак, можна додати методи для дерев розбору за допомогою вузлів розширення.

В першому рядку граматики повинно бути слово `grammar`, а потім один пробіл та назва граматики. Далі йде список правил, що визначають граматику; правило - це ім'я, за яким слідує <-, після чого слідує правило. Ім’ям правила може бути будь-яке дійсне ім'я змінної ASCII JavaScript. Перше правило в граматиці - `root`; документ повинен відповідати цьому правилу для правильного виконання розбору.

Рядки, що починаються з #, трактуються як коментарі.

Деталі типів виразів можна використовувати без граматичних правил

Наприклад, ось проста граматика, яка містить довільну послідовность цифр:

digits.peg

```
# A grammar file to be used with Canopy:
# https://www.npmjs.com/package/canopy
#
# Explanation and syntax reference: http://canopy.jcoglan.com/
#
# To build:
# `npm install -g canopy`
# `canopy digits.peg --lang javascript` (or java | python |
ruby)
grammar Digits
  digits  <-  [0-9]*
```

Граматичний файл перетворюється в модуль JavaScript за допомогою командного рядка Canopy:

```
$ canopy digits.peg
```

Правила можуть містити посилання на інші правила; саме це дозволяє парсерам PEG обробляти рекурсивні синтаксиси. Наприклад, ця граматика містить числа, які оточені довільною кількістю відповідних дужок - це неможливо при регулярних виразах.

parens.peg

```
grammar Parens
  value   <-  "(" value ")" / number
  number  <-  [0-9]+
```

Це згенерує аналізатор, який обробляє мову та спрямовує помилку на недійсний вхід:

```
var parens = require('./parens')

parens.parse('94')
   == { text: '94',
        offset: 0,
        elements:
         [ { text: '9', offset: 0, elements: [] },
           { text: '4', offset: 1, elements: [] } ] }

parens.parse('(94)')
   == { text: '(94)',
        offset: 0,
        elements:
         [ { text: '(', offset: 0, elements: [] },
           { text: '94', offset: 1, elements: [...] },
           { text: ')', offset: 3, elements: [] } ] }
```

```
parens.parse('(((94)')
Error: Line 1: expected ")"
(((94)
      ^
```

2.2. Співставлення рядків

Найосновніший тип вузла в PEG граматиках - це буквальна відповідність рядків. Ця граматика буде відповідати лише рядку "`I like parsers`" і ніякому іншому.

parsers.peg

```
grammar Parsers
  root  <-  "I like parsers"
```

Рядки розмежовані подвійними лапками. Рядковий вузол - це термінал - у ньому немає дочірніх вузлів.

```
require('./parsers').parse('I like parsers')
   == { text: 'I like parsers',
        offset: 0,
        elements: [] }

require('./parsers').parse('this is not valid')
Error: Line 1: expected "I like parsers"
this is not valid
^

require('./parsers').parse('I like PARSERS')
Error: Line 1: expected "I like parsers"
I like PARSERS
^
```

Як видно з останнього прикладу, рядки залежать від регістру. Ви можете створити нечутливі до регістру рядки, цитуючи рядок із зворотними посиланнями:

parsers.peg

```
grammar Parsers
  root  <-  `I like parsers`
```

Після цього він буде відповідати будь-якому регістру виводу:

```
require('./parsers').parse('I like PARSERS')
   == { text: 'I like PARSERS',
        offset: 0,
        elements: [] }
```

2.3. Символьні класи

Символьні класи працюють подібно до своїх аналогів з регулярних виразів, але насправді вони буквально компілюються в однакові регулярні вирази в аналізаторі.

Наприклад, ця граматика відповідає одному буквено-цифровому символу:

alphanum.peg

```
grammar Alphanum
  root  <-  [A-Za-z0-9]
```

Це дозволить розібрати будь-який символ, відповідний класу, і жоден інший:

```
require('./alphanum').parse('a')
   == { text: 'a', offset: 0, elements: [] }
```

```
require('./alphanum').parse('7')
   == { text: '7', offset: 0, elements: [] }

require('./alphanum').parse('!')
Error: Line 1: expected [A-Za-z0-9]
!
^
```

Існує спеціальний клас символів, позначений виразом `.(period)`. Він відповідає будь-якому символу `.`

anything.peg

```
grammar Anything
  root  <-  .
```

```
require('./anything').parse('a')
   == { text: 'a', offset: 0, elements: [] }
```

2.4. Опціональні вузли

Щоб зробити вузол опціональним, безпосередньо після нього використовується символ ?, без пробілу між вузлом та символом ?

Ось граматика, яка відповідає слову "`future`" або порожньому рядку:

future.peg

```
grammar Future
  root  <-  "future"?
```

А оце граматика, яка відповідає будь-якій цифрі або порожньому рядку:

digit.peg

```
grammar Digit
  root  <-  [0-9]?
```

Якщо опціональний вузол не знайдено на вході, повертається порожній синтаксичний вузол. Запам’ятайте, опціональний вузол або відповідає вузлу, або не відповідає взагалі нічому; інші вхідні дані призведуть до помилки.

```
require('./future').parse('future')
   == { text: 'future', offset: 0, elements: [] }

require('./future').parse('')
   == { text: '', offset: 0, elements: [] }

require('./future').parse('perfect')
Error: Line 1: expected "future"
perfect
^
```

2.5. Повторювані вузли

Canopy пропонує добре знайомі * та + оператори з регулярних виразів, що означають «нуль або більше» та «один чи більше» відповідно. Розміщення їх після вузла дозволяє вузлу виконати повторення потрібну кількість разів.

Ось граматика, яка задіює слово `badger` один чи більше разів:

badger.peg

```
grammar Badger
  root  <-  "badger"+
```

Коли повторення відповідає вводу, ви отримуєте один вузол, який повторює самого себе і один дочірній вузол для кожного екземпляра в повторенні. Якщо недостатньо повторень (наприклад, якщо немає хоча б одного використання правила + чи *) ви отримаєте помилку.

```
require('./badger').parse('badgerbadgerbadger')
   == { text: 'badgerbadgerbadger',
        offset: 0,
        elements:
         [ { text: 'badger', offset: 0, elements: [] },
           { text: 'badger', offset: 6, elements: [] },
           { text: 'badger', offset: 12, elements: [] } ] }

require('./badger').parse('bad')
Error: Line 1: expected "badger"
bad
^
```

2.6. Послідовності

Послідовність - це один чи більше вузлів, які слідують один за одним і розділені хоча б одним символом пробілу. Послідовність відповідає вхідним даним, якщо вхід містить відповідність для кожного з вузлів послідовності.

Наприклад ось граматика, яка відповідає опціональному слову з наступними двома потрібними словами:

hamlet.peg

```
grammar Hamlet
  root  <-  "not "? "to be"
```

Тут послідовність складається з двох вузлів: "not"? і "to be". Ось отриманий аналіз вхідних даних:

```
require('./hamlet').parse('to be')
   == { text: 'to be',
        offset: 0,
        elements:
         [ { text: '', offset: 0, elements: [] },
           { text: 'to be', offset: 0, elements: [] } ] }

require('./hamlet').parse('not to be')
   == { text: 'not to be',
        offset: 0,
        elements:
         [ { text: 'not ', offset: 0, elements: [] },
           { text: 'to be', offset: 4, elements: [] } ] }

require('./hamlet').parse('or not to be')
Error: Line 1: expected "to be"
or not to be
^
```

## Мічені вузли

Послідовності мають особливу властивість: їх дочірні вузли можна помітити. Можна явно додати мітку до будь-якого елемента в послідовності, а перехресні посилання неявно помічені іменем посилання. Наприклад, візьмемо такий приклад, який відповідає документам, схожим на `{'abc' => 123}`:

hash.peg

```
grammar Hash
  object  <-  "{" string " => " number:[0-9]+ "}"
  string  <-  "'" [^']* "'"
```

Правило об'єкта - це послідовність, що містить п'ять значень:

- “{”
- string

- “ => ”
- number : [0-9]+
- “}”

Рядковий вузол є посиланням на інше правило, а число: `[0-9]+` це мічений вираз, що відповідає одній або більше цифрам. Ці два значення створюють мічені вузли у виході:

```
tree = require('./hash').parse("{'foo' => 36}")

   == { text: '{\'foo\' => 36}',
        offset: 0,
        elements:
         [ { text: '{', offset: 0, elements: [] },
           { text: '\'foo\'', offset: 1, elements: [...] },
           { text: ' => ', offset: 6, elements: [] },
           { text: '36', offset: 10, elements: [...] },
           { text: '}', offset: 12, elements: [] } ],
        string:
         { text: '\'foo\'', offset: 1, elements: [...] },
        number:
         { text: '36', offset: 10, elements: [...] } }

tree.string.text
   == "'foo'"

tree.number.text
   == "36"
```

Тут ми бачимо, що `tree.string` - це те саме, що `tree.elements [1]`, а `tree.number` - те саме, що `tree.elements [3]`. Ці мітки полегшують навігацію по дереву та спрощують читання коду.

2.7. Випереджаючий пошук

Випереджаючий пошук, позначений `&` і `!` символами, можна використовувати для перевірки наступного фрагмента вводу, не задіяючи його. Вираз `& expr` відповідає фрагменту вводу, якщо він відповідає `expr`, а `! expr` відповідає, якщо він не відповідає expr.

Наприклад, це правило містить слово телевізор, за яким слідує двокрапка, але це ж правило не задіює двокрапку:

```
tv   <-  "television" &":"
```

Це правило відповідає будь-якій кількості символів, що не містять пробілів, спочатку перевіряючи наступний символ за допомогою негативного знаку пошуку та потім використовуючи його з оператором підстановки:

```
nonspace   <-  (!" " .)*
```

Оператори випереджаючого пошуку не повинні супроводжуватись рядками, вони можуть використовуватися з будь-яким іншим виразом розбору.

2.8. Впорядкований вибір

Вибір використовується для позначення місць, де є більше одного можливого синтаксису для поданих даних. Наприклад, наступна граматика відповідає або рядку "`hello`", або рядку "`world`":

choice.peg

```
grammar Choice
  root  <-  "hello" / "world"
```

Важливою особливістю граматики з PEG є те, що вона впорядкована. Це означає, що параметри пробуються впорядкуватись, і перший вибір, який веде до успішного розбору, зберігається. Якщо цей вибір призведе до помилки

синтаксичного аналізу при наступному вводі даних, аналізатор не буде намагатись виконувати будь-які інші варіанти, розбір просто не вдається.

Оператор вибору використовується більш вільно, ніж оператор послідовності, наприклад, наступна граматика відповідає або рядку "`DouglasAdams`", або рядку "`HitchhikersGuide`":

binding.peg

```
grammar Binding
  root  <-  "Douglas" "Adams" / "Hitchhikers" "Guide"
```

Якщо ви просто хочете, щоб вибір був між вузлами поруч з оператором `/`, потрібно скористатися дужками. Ця граматика відповідає словосполучення "`DouglasAdamsGuide`" або "`DouglasHitchhikersGuide`":

binding.peg

```
grammar Binding
  root  <-  "Douglas" ("Adams" / "Hitchhikers") "Guide"
```

2.9. Перехресні посилання

Можливість посилатися на інші правила аналізатора – це одна з ключових особливостей PEG-аналізаторів. Правило може посилатися на будь-яке інше правило граматики за назвою і це призводить до використання названого правила для відповідності наступному фрагменту введення. Ось спрощений приклад використання правил для відповідності адреси електронної пошти:

email.peg

```
grammar Email
  email     <-  username "@" host
  username  <-  [a-z]+ ("." [a-z]+)*
  host      <-  [a-z]+ "." ("com" / "co.uk" / "org" / "net")
```

```
require('./email').parse('bob@example.com')
   == { text: 'bob@example.com',
        offset: 0,
        elements:
         [ { text: 'bob', offset: 0, elements: [...] },
           { text: '@', offset: 3, elements: [] },
           { text: 'example.com', offset: 4, elements: [...] } ],
        username: { text: 'bob', offset: 0, elements: [...] },
        host:
         { text: 'example.com',
           offset: 4,
           elements: [...] } }
```

Як можна побачити у вищезгаданому дереві розбору, правила посилаються на правило `email`, та додають іменовані вузли, які називаються `username` та `host`, до дерева розбору. Це дає простіший спосіб перетнути дерево, ніж використовувати масив `elements`.

```
var tree = require('./email').parse('bob@example.com');
tree.username.text    == 'bob'
tree.host.text        == 'example.com'
```

Посилання дозволяють створювати рекурсивні відповідники, тому PEG можуть використовувати мови, які звичайні вирази використовувати не можуть. Ось граматика відповідності вкладених списків чисел:

### lists.peg

```
grammar Lists
  value   <-  list / number
  list    <-  "[" value ("," value)* "]"
  number  <-  [0-9]
```

Можемо проаналізувати рядок за допомогою цієї граматики та переглянути дерево, яке вона створює:

```
var tree = require('./lists').parse('[[1,2],3]')

tree.text
   == '[[1,2],3]'

tree.elements[1].text
   == '[1,2]'

tree.elements[1].elements[2].elements[0].elements[1].text
   == '2'
```

2.10. Створення дерев розбору

За замовчуванням аналізатор Canopy генерує дерево розбору, не потребуючи вказівок про те, як це зробити. Кожен вузол має `text`, `offset` та (можливо, порожній) список `elements`. Але також є можливість повідомити Canopy викликати визначені користувачем функцій, щоб створити дерево власноруч.

Скажімо, у нас є граматика, яка відповідає рядкам, які відображають імена списку чисел, наприклад `{'ints': [1,2,3]}`:

maps.peg

```
grammar Maps
  map     <-  "{" string ":" value "}"
  string  <-  "'" [^']* "'"
  value   <-  list / number
  list    <-  "[" value ("," value)* "]"
  number  <-  [0-9]+
```

Щоб змінити типи значень, які формує аналізатор кожного разу, коли він відповідає правилу, ми можемо дати імена функцій для виклику, які позначені як імена які містять в префіксі знак %:

maps.peg

```
grammar Maps
  map     <-  "{" string ":" value "}" %make_map
  string  <-  "'" [^']* "'" %make_string
  value   <-  list / number
  list    <-  "[" value ("," value)* "]" %make_list
  number  <-  [0-9]+ %make_number
```

Ці імена функцій називаються діями (actions). Після компіляції цього аналізатора ви можете використовувати його, передаючи об'єкт, який реалізує названі дії. Кожній функції передаються чотири аргументи:

- input: повний текст вхідного документа
- start: початкове зміщення тексту, що відповідає правилу
- end: кінцеве зміщення тексту, що відповідає правилу
- elements: масив значень, згенерованих підрегламентами правила

Наприклад, можна реалізувати дії для вищевказаного аналізатора, які переводять вхідний текст у JavaScript, без зміни структури:

```
var maps = require('./maps');

var actions = {
  make_map: function(input, start, end, elements) {
    var map = {};
    map[elements[1]] = elements[3];
    return map;
  },

  make_string: function(input, start, end, elements) {
    return elements[1].text;
```

```
  },

  make_list: function(input, start, end, elements) {
    var list = [elements[1]];
    elements[2].forEach(function(el) { list.push(el.value) });
    return list;
  },

  make_number: function(input, start, end, elements) {
    return parseInt(input.substring(start, end), 10);
  }
};

var result = maps.parse("{'ints':[1,2,3]}", {actions: actions});
console.log(result);
```

Ця програма видасть:

```
{ ints: [ 1, 2, 3 ] }
```

Аналізатор викликає ці дії замість того, щоб самостійно будувати вузли. Він передає аргументи (`input, start, end`), а не лише текст відповідності, оскільки це дозволяє скоротити час і пам'ять на створення підрядків, коли цього не потрібно; зауважте, як більшість наведених вище правил не використовують ці аргументи.

Оператор `%` прив'язується до виразів послідовностей, тобто в наступній граматиці вхідний `abc` буде викликати `make_alpha`, тоді як вхід `123` буде викликати `make_numeric`:

actions.peg

```
grammar Actions
  root  <-  "a" "b" "c" %make_alpha / "1" "2" "3" %make_numeric
```

Він використовується лише з виразами, які створюють нові вузли. Його не можна використовувати з виразами, які просто проходять через вузол, створений іншим правилом, таким як `?, /, & I!` операторами чи перехресними посиланнями. Він може використовуватися з послідовністю двох або більше виразів, яка містить правило

Функції дій викликаються під час запуску аналізатора, тому вони дозволяють виконувати код, поки вхідний текст ще обробляється.

## Додавання методів до вузлів

Замість того, щоб вказувати аналізатору, як створювати вузли, ви можете дозволити йому створити вузли, які він будує за замовчуванням за допомогою власних методів. Це робиться за допомогою анотування виразів розбору за типами. Тип - це будь-яке дійсне ім'я об’єкта JavaScript, наприклад: `Foo.Bar`, який оточений дужками. Коли вхід відповідає цьому виразу, згенерований синтаксичний вузол отримає методи із названого типу.

Візьмемо простий приклад: відповідність рядковій константі:

strings.peg

```
grammar Strings
  root  <-  "hello" <HelloNode>
var strings = require('./strings');

var types = {
  HelloNode: {
    upcase: function() {
      return this.text.toUpperCase();
    }
  }
};

var tree = strings.parse('hello', {types: types});
console.log(tree.upcase());
```

У граматиці сказано, що слово `hello` відповідає типу `HelloNode`. Потім у коді JavaScript ми передаємо об’єкт, який містить названі типи через параметр `type` і використовуємо парсер для обробки рядка.

Оскільки рядок відповідає нашому введеному правилу, він отримує методи з модуля `HelloNode` і ми можемо викликати ці методи в вузлі.

Запустимо цей сценарій:

```
$ node strings_test.js
HELLO
```

У синтаксисі граматики анотації прив’язуються до послідовностей. Тобто анотація типу може з’являтися лише в кінці виразу послідовності. На відміну від анотацій про дії, анотації типу можна використовувати для будь-якого виду вираження, не лише для тих, що створюють нові вузли.

Наприклад, наступний код означає, що вузол, що відповідає послідовності `"foo" "bar"`, буде доповнений методами `Extension`.

```
words.peg

grammar Words
  root  <-  first:"foo" second:"bar" <Extension>
```

Методи розширення мають доступ до міченого вузла з послідовності.

```
var words = require('./words');

var types = {
  Extension: {
    convert: function() {
      return this.first.text + this.second.text.toUpperCase();
    }
```

```
  }
};

words.parse('foobar', {types: types}).convert()
  == 'fooBAR'
```

Оскільки анотації типу пов'язуються з послідовностями, а не з виборами, наступні збігаються або з рядком "abc", який набирає тип Foo, або "123", який отримує тип Bar:

sequences.peg

```
grammar Choice
  root  <-  "a" "b" "c" <Foo> / "1" "2" "3" <Bar>
```

Якщо ви хочете, щоб усі гілки вибору були доповнені одним і тим же типом, вам потрібно скористатись дужками та поставити тип після цього.

choices.peg

```
grammar Choices
  root    <-  (alpha / beta) <Extension>
  alpha   <-  first:"a" second:"z"
  beta    <-  first:"j" second:"c"
```

```
var choices = require('./choices');

var types = {
  Extension: {
    convert: function() {
      return this.first.text + this.second.text.toUpperCase();
    }
  }
};
```

```
choices.parse('az', {types: types}).convert()
   == 'aZ'

choices.parse('jc', {types: types}).convert()
   == 'jC'
```

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Canopy – a parser compiler [Електронний ресурс] – Режим доступу: http://canopy.jcoglan.com
2. Canopy [Електронний ресурс] / GitHub – Режим доступу: https://github.com/jcoglan/canopy
3. Parsing [Електронний ресурс] / Вікіпедія – вільна енциклопедія. – Режим доступу: https://en.wikipedia.org/wiki/Parsing
4. PEG-парсеры [Електронний ресурс] / Habr. – Режим доступу: https://habr.com/ru/post/471860/