TKOM – Python testy jednostkowe

Dokumentacja projektu

Tomasz Załuska

Projekt wstępny

Treść zadania:

Napisać program, który analizuje kod w Pythonie (podzbiór Pythona) i tworzy proste testy jednostkowe (szkielet testów).

Podzbiór języka Python:

1. Instrukcje proste:

    a. Operacje arytmetyczne – program potrafi przetwarzać podstawowe operacje arytmetyczne, takie jak dodawanie, odejmowanie, mnożenie i dzielenie. Możliwe są także wyrażenia nawiasowe.
    b. Operacje logiczne oraz zawierania – program potrafi przetwarzać podstawowe operatory porównania, logiczne oraz zawierania, czyli (==, >, >=, !=, in, and, or).
    c. Komentarze – reprezentowany przez znak „#”, reszta linie nie jest analizowana przez parser.
    d. Przypisanie – lewa strona równania to zmienna o typie dynamicznym, natomiast prawa to przypisywana do niej wartość. Wartość może być liczbą, ciągiem znaków, lista lub obiektem klasy.
    e. Podstawowe funkcje wbudowane takiej jak print.

2. Instrukcje złożone:

    a. Pętla for – definiowana jest przez słowo kluczowe „for”, nazwę zmienną iterowanej w pętli, słowo kluczowe „in” oraz zakres. Zakres może być definiowany przez słowo kluczowe „range”.
    b. Pętla while – definiowana przez słowo kluczowe „while” oraz warunek logiczny zakończenia pętli.
    c. Warunek if – definiowany przez słowo kluczowe „if” oraz warunek logiczny.
    d. Wraunek else – definiowany tylko przez słowo kluczowe „else”. Musi być poprzedzony przez warunek if.

3. Klasy wraz z dziedziczeniem.

Obsługiwane słowa kluczowe:

And, class, def, else, for, if, in, or, print, return, while, self.

Składnia:

<instrukcja> ::= <komentarz> | „ „ | <instrukcja prosta> | <instrukcja złożona>

<komentarz>::= „#” <ciąg znaków>

<instrukcja prosta>::=<print> | <operacja arytmetyczna> | <przypisanie> | <warunek>

<print>::= ”print” „(„ <parametr print> „)”

<parametr>::= „ „ | <tekst> | <zmienna> | <operacja arytmetyczna>

<lista parametrów>::= <parametr> [„, ” <lista parametrów>]

<operacja arytmetyczna>::= <składnik> <operator> <składnik>

<składnik>::= <liczba> | <operacja arytmetyczna> | „(„ <operacja arytmetyczna> „)”

<tekst>::= „cudzysłów” <ciąg znaków> „cudzysłów ”

<ciąg znaków>::= „ „ | <znak> [<ciąg znaków>] | <cyfra [<ciąg znaków>] | <znak specjalny> [<ciąg znaków>]

<liczba>::= <cyfra> [<liczba>]

<cyfra>::= „0” | „1” | „2”| „3” | „4” | „5” | „6” | „7” | „8” | „9”

<znak>::= „a” | „b” | „c” | „d” | „e” | „f” | „g” | „h” | „i” | „j” | „k” | „l” | „m” | „n” | „o” | „r” | „s” | „t” | „u” | „w” | „x” | „y” | „z” | „A” | „B” | „C” | „D” | „E” | „F” | „G” | „H” | „I” | „J” | „K” | „L” | „M” | „N” | „O” | „P” | „R” | „S” | „T” | „U” | „W” | „X” | „Y” | „Z”

<znak specjalny::= „?” „/” „<” „>” „,” „.”

<przypisanie>::= <zmienna> „=” <wartość przypisywana>

<wartość przypisywana>::= <liczba> | <tekst> | <lista> | <zmienna> | <konstruktor klasy>

<lista>::= „[„ <element listy> [„ , ” <lista>] „]”

<element listy>::= <tekst> | <liczba>

<konstruktor klasy>::= <ciąg znaków>”(„ [<lista parametrów>] „)”

<instrukcja złożona>::= <wyrażenie> „:” | <definicja klasy>

<wyrażenie>::= <pętla for> | <pętla while> | <warunek if> | <warunek else>

<pętla for>::= „for” <zmienna> „in” <zakres pętli for>

<zakres pętli for>::= „range” „(„ <liczba> [, liczba] „]”

<pętla while>::= „while” <warunek>

<warunek if>::= „if” <warunek złożony>

<warunek else>::= „else” <warunek złożony>

<definicja klasy>::= „class” <ciąg znaków> „:” [„def __ init__(self, ” <lista zmiennych> „) :”

<warunek złożony>::= <warunek prosty> |<warunek prosty> [<operator łączenia warunków> <warunek złożony>]

<operator łączenia warunków>::= „and” | „or”

<warunek>::= <operand> <operator> <operand> | <zmienna>

<operand>::= <liczba> | <zmienna> | <tekst>

<operator>::= „!=” „==” „>” „<” „<=” „>=”

<zmienna>::= <znak początkowy> [<ciąg znaków>]

<lista zmiennych>::= <zmienna> [„, ” < lista zmiennych>]

<znak początkowy>::= <znak> | „_”

Budowa programu

1. Obsługa plików

Moduł programu odpowiedzialny za czytanie pliku i przekazywanie kolejnych znaków innym modułom programu. Będzie odpowiedzialny również za zapisywanie tekstu do pliku wyjściowego.

2. Analizator leksykalny

Moduł programu odpowiedzialny za tworzeniu kolejnych tokenów na podstawie kolejnych znaków z pliku wejściowego. Tokeny będą przekazywane do analizatora składniowego. Szczególnie ważne dla pracy moduł generatora testów będą tokeny klas i metod klas.

3. Analizator składniowy

Moduł programu odpowiedzialny za sprawdzenie czy tokeny otrzymywane od analizatora leksykalnego zgadzają się z gramatyką Pythona.

4. Analizator semantyczny

Moduł programu odpowiedzialny za sprawdzenie poprawności programu na poziomie znaczenia poszczególnych instrukcji oraz programu jako całości.

5. Generator testów jednostkowych

Moduł programu odpowiedzialny za generacje szablonów testów jednostkowych dla metod klas. Moduł ten będzie szukał metod klas, dla których należy utworzyć szablony testów na podstawie drzewa rozbioru. Następnie dla każdej metody zostanie utworzona klasa, w której będzie szablon jednego scenariuszu testu. Będą również generowane szablony testów z parametrami. Moduł ten będzie miał dostęp do drzewa składniowego.

Przykład:

Plik wejściowy:

```
class Generator:
	def generate(self, number_of_shares, lower_price_limit, upper_price_limit):
	#ciało metody
	def generate_decreasing(self,number_of_shares, p):
	#ciało metody
```

Plik Wyjściowy:

```
import unittest
import <plik wejściowy>

class test_class_Generator_methode_generate(unittest.TestCase):
	def test_1(self):
	  self.fail("Not yet implemented")
	def test_parameters_1(self, number_of_shares, lower_price_limit,
	upper_price_limit):
	  self.fail("Not yet implemented")

class test_class_Generator_methode_generate_decreasing(unittest.TestCase):
	def test_1(self):
	  self.fail("Not yet implemented")
	def test_parameters_1(self, number_of_shares, lower_price_limit,
	upper_price_limit):
	  self.fail("Not yet implemented")
```

6. Obsługa błędów

Moduł programu odpowiedzialny za odbieranie informacji od reszty modułów o wystąpieniu błędu. W przypadku wystąpienia błędu analiza zostanie przerwana i zostanie wyświetlony odpowiedni komunikat błędu.

Program

Program będzie aplikacja konsolową napisaną w języku wysokiego poziomu (C++ lub Java lub Python). Na wejściu będzie otrzymywał plik do analizy, a na wyjściu będzie generowany plik wyjściowy zawierający szablony testów jednostkowych. W celu testowania zostaną napisane testy jednostkowe.

Struktury danych

1. Tablica tokenów

   Tablica tokenów generowana na podstawie pliku wejściowego przez analizator leksykalny. W tablicy będą przechowywane informacje jaki jest charakter tokenu (czy jest to zmienna, czy stała), nazwa lub wartość tokenu oraz dodatkowe informacje (w razie potrzeby).

2. Drzewo składniowe

Drzewo tokenów tworzone na podstawie tablicy symbolów przez analizator składniowy w celu sprawdzenia zdefioniowanej gramatyki. Węzeł drzewa będzie zawierał informacje o tokenie jaki reprezentuje.

Przykłady testów

1. Wejście:

```
class Osoba:
    def __init__(self, imie, nazwisko, wiek):
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek = wiek
    def przedstaw_sie(self):
        print(f"Jestem {self.imie} {self.nazwisko}. Mam {self.wiek} lat.")
```

Wyjście:

```
import unittest
import <plik wjeściowy>

class test_class_Osoba_methode_przedstaw_sie(unittest.TestCase):
    def test_1(self):
        self.fail("Not yet implemented")
    def test_parameters_1(self, number_of_shares, lower_price_limit,
upper_price_limit):
        self.fail("Not yet implemented")
```

2. Wejście:

```
1       name = "Jim
```

Wyjście:

Błąd składniowy (wiersz nr 1)

Dokumentacja końcowa

Wprowadzone zmiany względem projektu wstępnego:

1. Brak implementacji obsługi komentarzy przez parser.
2. Brak implementacji obsługi pętli for oraz funkcji range().

Opis struktury porojektu

1. ut_lexer.py
   Plik ten zawiera klasę Lexer, która implementuję lekser. Klasa jest odpowiedzialna za przetwarzanie źródła danych i wyprodukowanie obiektów Token z biblioteki ply.lex.

2. parse_model.py

   Plik ten zawiera zbiór klas (produkcji, które mogą wystąpić w programie), na których podstawie jest budowane drzewo składniowe programu.

3. ut_parser.py

   Plik zawiera opis zależności między produkcjami. Na podstawie tych zależności jest budowany parser przy zastosowaniu biblioteki ply.yacc.

4. unittest_generator.py

   Plik implementuje generator testów jednostkowy, który przechodząc po drzewie składniowym wyszukuje wszystkich klas i dla każdej metody(oprócz konstruktora) w tej klasie tworzy szablon testu jednostkowego.

5. main.py
   Plik z implementacją głównej funkcji programu, która parsuje parametry wywołania. Na ich podstawie wywołuje parser oraz generator testów jednostkowych.

6. lex_test.py

   Implementacja testów jednostkowych dla klasy Lexer.

7. parser_test.py
   Implementacja testu parsera polegająca na sprawdzeniu poprawności parsowania pewnej klasy.

8. test_code.py oraz out.py
   Plik test_code.py to przykładowy plik wejściowy dla programu, a out.py to plik wyjściowy dla tego przykładowego pliku wejściowego.

Testowanie

W celu sprawdzenia poprawności działania zostały napisane testy jednostkowe, dla leksera i parsera. Przeprowadzone zostały również testy funkcjonalne, dla przykładów testowych podanych w dokumentacji.

Sposób wykorzystania

Aby uruchomić generator testów jednostkowych należy wykonać w konsoli polecenie.

```
$ python main.py -i <plik wejściowy> -o ,plik wyjściowy>
```

Lub

```
$ python main.py -i <plik wejściowy>
```

Biblioteki pythona

Do poprawnego działania programu należy pobrać bibliotekę pythona o nazwie „ply”.