

Dokumentacja wstępna

Autor: Artur Wyróżębski

Opis projektu

Napisać program, który analizuje kod w podzbiorze Python i znajduje nieużywane elementy programu: zmienne, stałe, klasy, funkcje, metody, konstruktory, etc.

Założenia

Wszelkie elementy programu Python, aby mogły zostać poprawnie wykorzystane, muszą być w tej samej lub nadrzędnej przestrzeni nazw. Jeżeli nazwa elementu zostanie użyta w określonej przestrzeni nazw, a potem w nadrzędnej przestrzeni nazw zostanie zdefiniowana ta sama nazwa, to muszą być one traktowane jako niezwiązane ze sobą.

Zawartość podzbioru języka Python

Analizowany będzie język Python w wersji 3.7.

Dopuszczalne są:

funkcje, klasy, instrukcje warunkowe, pętle, zmienne, moduły, komentarze, standardowe struktury danych (list, range, dict, set), adnotacje typu.

Wymagania funkcjonalne

- Odczytywanie i analiza kodu źródłowego języka Python zapisanego w plikach tekstowych oraz ze standardowego wejścia.
- Branie pod uwagę adnotacji typów zmiennych podczas analizy operacji na tych zmiennych i wyświetlenie błędu, jeżeli typy są ze sobą niekompatybilne.

Wymagania niefunkcjonalne

- Komunikat o nieużywanym elemencie powinien wskazywać miejsce w kodzie, w którym znajduje się ten element oraz przedstawiać nazwę tego elementu i jego rodzaj (klasa, zmienna, funkcja etc.).
- Komunikat o błędach analizy powinien być prosty i przejrzysty.
- Poprawne ukazanie wszystkich nieużywanych elementów kodu źródłowego na standardowym wyjściu.

Środowisko i technologie

Do utworzenia programu wykorzystany zostanie język Java w wersji 14.

Do tworzenia testów jednostkowych wykorzystania zostanie biblioteka JUnit.

Projekt jest aplikacją konsolową.

Sposób uruchomienia

Program jest aplikacją konsolową, którą uruchamia się poprzez wywołanie z parametrem reprezentującym ścieżkę do pliku z kodem źródłowym. Program może być również uruchomiony bez jakichkolwiek argumentów, wtedy przyjmuje on kod źródłowy ze standardowego wejścia.

Nazwa programu to 'linter'.

Pliki do analizy muszą kończyć się rozszerzeniem .py.

Przykłady uruchomienia:

./linter plik1.py

./linter

Wyjście:

Format wyświetlenia nieużywanego elementu:

Unused : wiersz;kolumna (pozycja w źródle) : nazwa : rodzaj elementu

Rodzaje elementów:

variable, function, class, method, module

Przykłady:

Unused : 3;5 : suma : variable

Unused : 10;3 : fun : function

Format wyświetlania błędów:

Error : wiersz;kolumna : rodzaj błędu

Przykłady:

Error : 10;5 : Incompatible types

Error : 15;3 : Undefined name

Zdefiniowane tokeny

'class', 'def', 'import', 'as', '+=', '-=', '*=', '/=', '%=', '**=', '//=', 'return',
'if', 'else', 'elif', 'break', 'continue', 'pass', '<', '>', '==', '>=', '<=', '<>',
'!=', 'in', 'not', 'is', ':', '[', ']', '(', ')', '{', '}', 'and', 'or'

Gramatyka

single_input = 'NEWLINE' | simple_stmt | compound_stmt 'NEWLINE'

file_input = ('NEWLINE' | stmt)* 'ENDMARKER'

stmt = simple_stmt | compound_stmt

simple_stmt = (assign_stmt | pass_stmt | flow_stmt | import_stmt) 'NEWLINE'

assign_stmt = var_name (annassign | augassign)

annassign = '=' (var_name '=')* expr

augassign = ('+=' | '-=' | '*=' | '/=' | '%=' | '**=' | '//=') expr

pass_stmt = 'pass'

flow_stmt = break_stmt | continue_stmt | return_stmt

break_stmt = 'break'

continue_stmt = 'continue'

return_stmt = 'return' [test]

```

import_stmt = import_name | import_from
import_name = "import" dotted_as_names
dotted_as_names = dotted_as_name (',' dotted_as_name)*
dotted_as_name = name ('.' name)* ['as' name]

compound_stmt = if_stmt | while_stmt | for_stmt | func_def | class_def
if_stmt = "if" test ':' suite ('elif' test ':' suite)* ["else" ':' suite]
while_stmt = "while" test ':' suite ['else' ':' suite]
for_stmt = "for" names_list "in" atom_expr ':' suite
suite = simple_stmt | "NEWLINE" "INDENT" {stmt} "DEINDENT"
names_list = name (',' name)*

test = or_test
or_test = and_test ("or" and_test)*
and_test = not_test ("and" not_test)*
not_test = "not" not_test | comparison
comparison = expr (comp_op expr)*
comp_op = "<" | ">" | "==" | ">=" | "<=" | "<>" | "!=" | "in" | "not in" | "is" | "is not"
expr = term (('+' | '-' term)*)
term = factor (('*' | '/' | '%' | '//') factor)*
factor = ('+' | '-' | '~') factor | power
power = atom_expr ['**' factor]
atom_expr = atom trailer*
atom = ("(" test_list ")") | "[" test_list "]" | "{" make_dict_or_set "}" |
| name | number | "None" | "True" | "False"

test_list = test (',' test)* [',']
make_dict_or_set = ((test ':' test) (',' (test ':' test)))* [','] | test_list

trailer = ("[" passed_arg ")") | "[" subscriptlist "]" | "." name
passed_arg = test ["=" test]
subscriptlist = subscript (',' subscript)* [',']
subscript = test | [test] ':' [test] [':' test]

class_def = "class" name [("[" arglist ")")] ':' suite

func_def = "def" name parameters ["->" name] ':' suite
parameters = ("[" arguments ")")
arguments = argument (',' argument)*
argument = test [':' test] ["=" test]

name = letter (digit | letter)*
number = "1".."9" digit*
letter = "a".."z" | "A".."Z" | "_"
digit = "0".."9"

```

Moduły

Program składa się z czterech głównych modułów i kilku modułów pomocniczych.

Analiza leksykalna

Odpowiedzialność za tworzenie tokenów. Odczyt ze strumienia odbywa się znak po znaku i po utworzeniu poprawnego tokenu tenże token przesyłany jest do analizatora składniowego. Wspomagającymi modułami są moduł odczytu strumienia wejściowego, moduł obsługi błędów i moduł akceptowalnych tokenów.

Analiza składniowa

Odpowiedzialność za stwierdzenie poprawności gramatycznej i tworzenie drzewa rozbioru. Wspomagającym modułem jest moduł obsługi błędów.

Analiza semantyczna

Odpowiedzialność za stwierdzenie poprawności utworzonego drzewa składniowego np. sprawdzenie czy identyfikatory argumentów funkcji się nie powtarzają.

Dodaje on identyfikatory do drzewa przestrzeni nazw.

Aby sprawdzić poprawność adnotacji typu, gdzie podana jest klasa jako typ, analizowana jest najpierw tablica akceptowalnych tokenów, a potem rozbudowywane drzewo przestrzeni nazw. Odpowiedzialność znalezienia klasy oddelegowywana jest modułowi analizy użyteczności identyfikatorów, który zwraca informację czy udało się znaleźć, czy nie. Wspomagającym modułem jest moduł obsługi błędów.

Analiza użyteczności identyfikatorów

Odpowiedzialność za znalezienie identyfikatorów, które nie są używane w programie.

Analizuje on drzewo przestrzeni nazw i dodaje on do struktury danych (zbioru) identyfikatory, które nie zostały (jeszcze) wykorzystane. Jeżeli identyfikator pojawi się drugi raz w przestrzeni nazw, to zostanie on usunięty z tego zbioru.

Zwraca do modułu wyjścia końcowy stan zbioru nieużytych identyfikatorów.

Wspomagającym modułem jest moduł obsługi błędów.

Moduły pomocnicze

Wejście – pobieranie znaków ze strumienia

Obsługa błędów – odpowiada za zwrócenie do modułu wyjścia odpowiedniego tekstu błędu.

Wyjście – zwraca użytkownikowi wyniki poprawnego działania programu oraz informacje z modułu obsługi błędów.

Zwracane błędy do modułu obsługi błędów

Moduł wejścia: związane z pobieraniem znaków, uruchamianiem programu.

Moduł analizy leksykalnej: związane z tworzeniem tokenów np. nieistniejący token.

Moduł analizy składniowej: związane ze składnią i tworzeniem drzewa rozbioru.

Moduł analizy semantycznej: związane z analizą drzewa rozbioru i tworzeniem drzewa przestrzeni nazw. Zwraca zwłaszcza błędy związane z niepoprawnymi operacjami związanymi z typami.

Moduł analizy użyteczności identyfikatorów: związane z analizą drzewa przestrzeni nazw np. niepoprawne zbudowanie drzewa przez moduł analizy semantycznej.

Struktury danych

Tablica akceptowalnych tokenów:

Zawiera powiązania słów lub znaków kluczowych z odpowiednią klasą słowa lub znaku kluczowego. Wartości kluczowe w danej klasie są enumerowane.

Drzewo przestrzeni nazw:

Drzewo przestrzeni nazw jest strukturą hierarchiczną zawierającą identyfikatory.

Odpowiedzialność za stworzenie tej struktury ponosi moduł analizy semantycznej, natomiast moduł analizy użyteczności identyfikatorów odpowiada za jego interpretację.

Węzłami tego drzewa są identyfikatory oraz podrzędne przestrzenie nazw w których mogą znajdować się kolejne identyfikatory oraz przestrzenie nazw.

Jest on strukturą uporządkowaną, to znaczy kolejność wstawiania elementów do drzewa jest istotna dla modułu analizy użyteczności identyfikatorów.

Analizator drzewa, aby sprawdzić czy identyfikator się pojawia więcej niż jeden raz, analizuje drzewo od danego węzła identyfikatora poruszając się tylko wstecz. Nie może on zejść do podrzędnego węzła przestrzeni nazw względem tego, w który obecnie analizuje, ale może zmienić przestrzeń nazw do nadrzędnej. Jedyne dopuszczalne „zejście” do podrzędnego jest wtedy, gdy występowała przed identyfikatorem kropka przed którą podany był moduł lub klasa. W tym wypadku należy przeanalizować przestrzenie nazw należące do tych elementów. Aby było to możliwe, w węzłach musi być jawnie podana informacja jaki ma identyfikator nadrzędna klasa lub moduł i musi być również uwzględnione to, że klasy mogą być zagnieżdżone w innej klasie. Aby więc znaleźć właściwy element, należy przeszukiwać od najbardziej nadrzędnego identyfikatora (przed pierwszą kropką) do najbardziej podrzędnego (po ostatniej kropce), aby schodzić do przestrzeni nazw.

Niestety w języku Python identyfikator zmiennej klasy może być zdefiniowany w dowolnej funkcji należącej do tej klasy po słowie kluczowym `‘self’`. Dla uproszczenia jest analizowana jedynie funkcja `__init__`, aby znaleźć identyfikator zmiennej gdy zajdzie taka potrzeba z zewnątrz.

Elementy węzła przestrzeni nazw:

Tablica zawierająca wskazania na inne węzły

Typ węzła (class, module, function, variable, unnamed)

Lista jednokierunkowa nazw identyfikatorów (możliwość pustej listy jeżeli typ unnamed)

Licznik użycia identyfikatora

Do typu `‘unnamed’` należą przestrzenie nazw wewnątrz bloku pętli lub warunku.

Licznik użycia identyfikatora zwiększa się w przypadku znalezienia tej samej nazwy węzła w drzewie (jeżeli nazwa niepusta). Wtedy zwiększa się wartość u węzła od którego zaczęto przeszukiwanie, jak i u tego znalezionego odpowiadającego pierwotnemu. Jest to konieczne, bo węzły zmiennych klas mogą nie wiedzieć o zewnętrznych odwołaniach do nich.

Identyfikator jest dodany do zbioru nieużytych identyfikatorów, jeżeli licznik użycia identyfikatora wynosi 0. Identyfikatory mogą zostać również usunięte z tego zbioru, gdy licznik znalezionego identyfikatora odpowiadającego sprawdzanemu zostanie zwiększony z 0.

Testowanie

Napisane zostaną testy jednostkowe oraz integracyjne projektu sprawdzające poprawność implementacji.

Przykładowe programy i poprawne wyjścia:

Program 1:

```
var = 3
def fun():
    if var == 3:
        var += 3
    else:
        x = 5
```

Wyjście analizatora:

Unused : 2;4 : fun : function

Unused : 6;8 : x : variable

Program 2:

```
def fun(x: List[int], y: int, z: float):
    y += x
```

Wyjście analizatora:

Error : 2;5 : Incompatible types

Unused : 1;30 : z : variable

Program 3:

```
def fun(x: int, y) -> int:
    return x+y
x = 3
y = 5
fun(y)
```

Wyjście analizatora:

Unused : 3;1 : x : variable

Program 4:

```
while(True):
def fun():
    x = 5
print("Hello world")
```

Wyjście analizatora:

Error : 1;1 : Undefined name

Program 5:

```
while(True):  
    x = 6  
    def _fun():  
        x += 5  
    print("Hello world")
```

Wyjście analizatora:

Unused : 3;8 : _fun : function