Dokumentacja

Autor: Artur Wyrozębski

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, standardowe struktury danych (list, tuple), adnotacje typu, importowanie.

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:
./java linter plik1.py
./java linter

Wyjście:

Format wyświetlenia nieużywanego elementu:
Unused ‘’rodzaj elementu’’: (pozycja w źródle) : nazwa : wiersz;kolumna

Rodzaje elementów:
variable, function, class

Przykłady:
Unused variable : x : 4;11
Unused class : Klasa : 9;7

Format wyświetlania błędów:
Error : Nazwa modułu : rodzaj błędu : wiersz;kolumna

Przykłady:
Error : Parser : Bad syntax : 18;5
Error : SemanticsAnalizer : Uninitialized variable : 18;1

Zdefiniowane tokeny

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

Gramatyka

single_imput = ‘’NEWLINE’’ | simple_stmt ‘’NEWLINE’’ | compound_stmt
simple_stmt = assign_stmt | pass_stmt | flow_stmt | import_stmt
assign_stmt = test [annasign | augassign]
annasign = (‘’=’’ 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’’ module [‘’as’’ name] (‘’,’’ module [‘’as’’ name])*
import_from = ‘’from’’ module ‘’import’’ name [‘’as’’ name] (‘’,’’ name [‘’as’’ name])*
module = name (‘’.’’ name)*

compound_stmt = if_stmt | while_stmt | for_stmt | func_def | class_def

if_stmt = ‘’if’’ test ‘’:’’ ‘’NEWLINE’’ suite (‘’elif’’ test ‘’:’’ suite)* [‘’else’’ ‘’:’’ ‘’NEWLINE’’ suite]
while_stmt = ''while'' test ‘':'' ‘’NEWLINE’’ suite [''else'' '':'' ‘’NEWLINE’’ suite]
for_stmt = ‘'for'' names_list ''in'' atom_expr '':'' ‘’NEWLINE’’ suite
suite = ‘’INDENT’’ {compound_stmt | simple_stmt ‘’NEWLINE’’ } ‘’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 ‘’]’’ | name | number | ‘’None’’ | ‘’True’’ | ‘’False’’)
test_list = test (‘','' test)*
trailer = ‘’(‘’ test* ‘’)’’ | ‘’.’’ name

class_def = ‘’class’’ name [‘’(‘’ [arglist] ‘’)’’] ‘’:’’ suite
func_def = ‘'def'' name parameters [''->'' name] ‘':'' suite
parameters = ‘’(‘’ [arguments] ‘’)’’
arguments = argument (‘’,’’ argument)*
argument = 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 lub czy zmienna nie została już utworzona.
Dodaje on zmienne, funkcje i klasy do przestrzeni nazw i zlicza liczbę odniesień do poszczególnych poprzednio wymienionych elementów.
Wspomagającym modułem jest moduł obsługi błędów.

Analiza użyteczności identyfikatorów
Odpowiedzialność za znalezienie identyfikatorów, które nie są używane w programie. Analizuje on globalną przestrzeń nazw oraz wszelkie lokalne, które powstały podczas analizy kodu. Jeżeli zauważy, że liczba odniesień do danego elementu przestrzeni nazw wynosi 0, to dodaje on ten element do swojej listy elementów nieużywanych. Zawartość tejże listy zostaje potem wyświetlona na ekranie.
Zwraca użytkownikowi wyniki poprawnego działania programu.

Moduły pomocnicze
Wejście – pobieranie znaków ze strumienia
Obsługa błędów – odpowiada za wyświetlenie odpowiedniego tekstu błędu. Wystąpienie błędu powoduje zatrzymanie programu.

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 przestrzeni nazw. Zwraca zwłaszcza błędy związane z niepoprawnymi operacjami związanymi z typami.

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.

Przestrzeń nazw

Program ma do dyspozycji przestrzeń nazw, która jest stosem. Na samym dnie stosu znajduje się globalna przestrzeń nazw, gdzie składowane są elementy globalne. Tymi elementami są zmienne, funkcje i klasy. Każde wyrażenie złożone tworzy swoją lokalną przestrzeń nazw, gdzie mogą znajdować się wszystkie elementy zdefiniowane w bloku danego wyrażenia złożonego. Jeżeli zakończy się jego blok, to odpowiadająca przestrzeń nazw jest przenoszona do listy wycofanych przestrzeni nazw. Jest ona konieczna do późniejszego znalezienia elementów niewykorzystywanych.

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 function : fun : 2;5
Unused variable : x : 6;3

Program 2:

```
def fun(x: list, y: int, z: float):
        y += x
```

Wyjście analizatora:
Error : SemanticsAnalizer : Incomaptible variable types : 2;2

Program 3:

```
def fun(x: int, y) -> int:
        return x+y
x = 3
y = 5
fun(1, y)
```

Wyjście analizatora:
Unused variable : x : 3;1

Program 4:

```
while? True:
      pass
```

Wyjście analizatora:
Error : Lexer : Bad token : 1;6

Program 5:

```
while True:
        x = 6
        def _fun():
                x += 5
```

Wyjście analizatora:
Unused function : _fun : 3;6