Dokumentacja wstępna

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.

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_imput = ‘’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 (annasign | augassign)
annasign = ‘’=’’ (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 zmiennych się nie powtarzają w tej samej przestrzeni nazw. Dodaje on identyfikatory do drzewa przestrzeni nazw i jeżeli to zmienna, to dodaje do identyfikatora znacznik, czy nastąpiła operacja przypisania 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.

Moduły pomocnicze

Wejście – pobieranie znaków ze strumienia
Obsługa błędów – odpowiada za zwrócenie użytkownikowi odpowiedniego błędu
Wyjście – zwraca użytkownikowi wyniki poprawnego działania programu
Tablica akceptowalnych tokenów – struktura zawierająca wszystkie obsługiwane tokeny.
Drzewo przestrzeni nazw – struktura danych zawierająca identyfikatory.

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 : 3;4 : fun : function
Unused : 7;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 : 4;1 : x : variable

Program 4:

```
whle(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 : 2;4 : x : variable
Unused : 4;8 : x : variable
Unused : 3;8 : _fun : function