PyGDB — analiza strukture izvornog koda

Projekt iz kolegija Napredne baze podataka

Luka Šimek

Zagreb, 14. svibnja 2024.*

Sadržaj

*datum zadnje izmjene: 8. lipnja 2024.

1 Uvod

1.1 Motivacija

Suvremeni softverski paketi često u sebi sadrže nepregledno veliku količinu raznih elemenata, od modula, klasa i funkcija do samog broja linija koda. Kao korisnicima, često nam se korisno malo bolje upoznati s arhitekturom paketa s kojim radimo, bilo da je zbog što kvalitetnijeg korištenja njegovih funkcionalnosti ili debuggiranja vlastitog koda.

S tim ciljem, pogodno bi bilo imati način za analizirati strukturu nekog paketa vizualno i potencijalno iz ptičje perspektive. Također, htjeli bismo da ta analiza bude fleksibilna — kao što smo naveli, cjelokupni sadržaj paketa može biti vrlo nepregledan, stoga želimo istovremeno biti u mogućnosti vizualizirati različite module u paketu, ali i podatke o tek nekolicini konkretnih funkcija.

Primijetimo da se spomenuta struktura prirodno preslikava na graf. Za vrhove grafa uzimamo različite objekte (potpakete, module, klase, funkcije i ostale varijable), dok su bridovi dani njihovim raznim odnosima (klasa je definirana u modulu, funkcija je metoda klase, jedna klasa naslijeđuje od druge itd.)

Naši zahtjevi — sakupljanje velike količine podataka o paketu ili više njih, fleksibilno dohvaćanje raznih dijelova tih podataka i modeliranje preko grafa navode nas na korištenje grafovske baze podataka. Konkretno, u ovom projektu koristit ćemo Neo4j. Jedan od razloga za to je da je riječ o trenutno najpoznatijem sustavu za upravljanje grafovskim bazama podataka. Cypher, njegov jezik za upite, sličan je poznatom SQL-u u sinktaksi, ali i sam po sebi vrlo kvalitetan i pogodan za upite nad grafovskim bazama. Dodatno, Neo4j Desktop u sebi već ima ugrađeno korisničko sučelje i mogućnosti vizualizacije rezultata upita.

1.2 Python i njegove značajke

Iako različiti programski jezici mogu dijeliti implicitnu shemu takve grafovske baze, algoritam za transformaciju programskog koda u graf nužno će ovisiti o programskom jeziku u kojemu je kod pisan. Ni zajednička shema nije nužna. Primjerice, već smo nekoliko puta spomenuli klase, no klase nisu dio svih programskih jezika.

U ovom projektu koristit ćemo Python i baviti se paketima pisanima, barem prvenstveno, u Pythonu. Jedan razlog svakako je u popularnosti jezika — Python je popularan jezik općenite uporabe i koristi se u velikom broju otvorenih paketa. Posebno spomenimo cijeli *data science* ekosustav u kojemu je Python zauzeo jednu od vodećih uloga u izradi biblioteka za znanstveno računanje, obradu i vizualizaciju podataka, statistiku i statističko učenje.

Činjenica da je Python dinamički tipiziran predstavlja izazov, ali i povećava smisao ovakvog projekta. U Pythonu ne postoje varijable u tipičnom smislu, već postoje samo *imena* koja možemo pridružiti objektima. To pridruživanje je potpuno fleksibilno. Primjerice, sljedeći kod je sasvim pravilan:

```
class A:
  pass

a = A()
A = 3
```

Dakle, dinamički je određeno na koji se objekt odnosti oznaka `A`. U ovom projektu zadržavamo se na statičkoj analizi koda — dinamička analiza bila bi skuplja vremenski i memorijski te bi bila dosta kompleksnija. Dodatno, bila bi ovisna o ispravno postavljenom i funkcionalnom softverskom okruženju. Zauzvrat, naši rezultati ne moraju uvijek biti pouzdani, ali se situacija kao iznad gotovo nikad neće naći u praksi, među poznatim i kvalitetno napravljenim paketima.

Treći razlog za Python je njegova opsežna standardna biblioteka koja će nam omogućiti interakciju sa strukturama podataka koje inače koristi samo kompajler. Više o tome ćemo reći u sljedećem poglavlju.

2 Alati

2.1 Apstraktna sintaksna stabla

Apstraktno sintaksno stablo (eng. *abstract syntax tree*, *AST*) je struktura podataka koja predstavlja strukturu programskog koda u nekom programskom jeziku. Vrhovi tog stabla odgovaraju sintaktičkim konstruktima u jeziku. Primjerice, vrh može odgovarati binarnoj operaciji. U tom slučaju lijevo i desno podstablo odgovaraju operandima, koji, rekurzivno, sami mogu imati više binarnih operacija u sebi. Vrh može odgovarati for-petlji — tada su sva djeca podstabla koja odgovaraju konstruktima unutar bloka određenog for-petljom. Ostali važni podaci, kao što su varijabla i granice iteracije, mogu biti spremljeni kao atributi tog objekta ili ponovo kao djeca glavnog vrha.

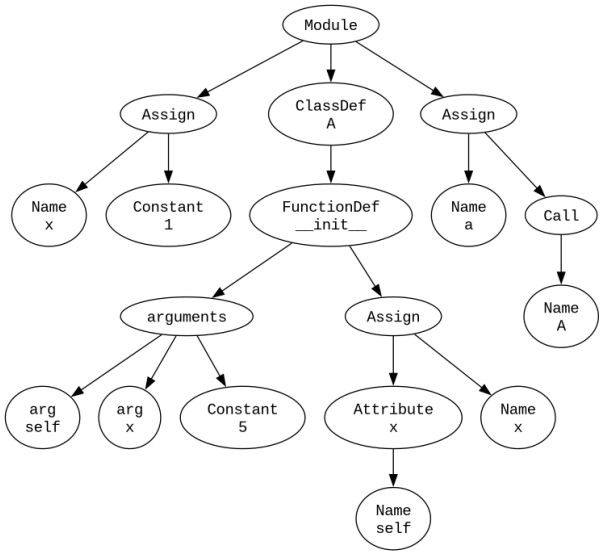
Apstraktna sintaksna stabla su međukorak prilikom kompilacije programskog koda u većini modernih programskih jezika. Tako i Pythonov kompajler stvara AST prije eventualnog prijevoda u *bytecode*. Apstraktna sintaksna stabla su dio samog kompajlera, ali modul `ast.py` (službena dokumentacija pod [2], podrobnija dokumentacija pod [5]) u standardnoj biblioteci služi kao omotač koji nudi AST kao objekt u Pythonu s nekoliko pomoćnih funkcija za njihovu obradu.

Postojeće funkcionalnosti modula `ast.py` dozvoljavaju pregled stabla u formatu sličnom JSON-u (`ast.dump`), ali pomoću paketa Graphviz ih možemo vizualizirati grafički. Razmotrimo sljedeći isječak koda:

```
x = 1
class A:
    def __init__(self, x=5):
        self.x = x

a = A()
```

Odgovarajuće apstraktno sintaksno stablo dano je na slici 2.1. Tekst u čvorovima odgovara različitim klasama u modulu, potklasama bazne klase `ast.AST` (npr. `ast.ClassDef` odgovara definiranju klase), a dodatne informacije, kao što je ime objekta ili vrijednost konstante, dane su ispod. Najčešće, čvorovi imaju još relevantnih atributa, ali na slici nisu prikazani u svrhu preglednosti.



Slika 1: Primjer AST-a

2.2 Tablice simbola

Tablica simbola (eng. *symbol table*) je još jedna struktura podataka koju koriste kompajleri. Generira se nakon AST-a, te u Pythonovom slučaju direktno prethodi konačnoj kompilaciji. Kao i AST, moguć joj je pristup kao objektu u Pythonu preko modula `symtable.py` (službena dokumentacija pod [9]) iz standardne biblioteke.

Tablica simbola sadrži sve simbole u kodu, a posebno se pritom osvrće na njihove opsege (*scope*-ove) i eventualno neke dodatne atribute. Pritom, glavna tablica sadrži globalni nazivni prostor (*namespace*), a može sadržavati i podtablice sa vlastitim, lokalnim nazivnim prostorima. U Pythonu, to je slučaj za nazivne prostore klasa i funkcija, ali ne, za razliku od drugih jezika, for-petlji ili sličnih blokova koje nemaju vlastiti nazivni prostor.

Budući da se tablica generira iz samog AST-a, sve što možemo napraviti s tablicama simbola možemo i sa samim AST-om. Ipak, uporaba modula iz standardne biblioteke smanjuje mogućnost greške i osigurava usklađenost s Pythonovim kompajlerom. Za primjer toga što je sve dostupno u Pythonu, čitatelj može pogledati primjer ispisa u bilježnici

`example/symtable_example.ipynb`

u repozitoriju projekta [1].

3 Konstrukcija grafa

Apstraktno sintaksno stablo je i samo graf, ali se značajno razlikuje od grafa kakvog smo opisali u 1. Primijetimo, još jednom, da slijed vrhova i bridova u AST-u odgovara slijedu samog koda. Imena i atributi definirani u njemu su smisleni Pythonovom interpreteru prilikom dinamičkog izvođenja programa, ali da bismo sami mogli raspoznati imena, njihove uloge i prodrijeti kroz ljudskom oku ponekad konvoluiranu strukturu AST-a, potreban je dodatni rad. Zbog toga želimo AST transformirati u prikladniji oblik grafa, koji ćemo kasnije prenijeti u grafovsku bazu podataka, a koja će omogućiti vizualizaciju grafa i izvršavanje upita.

Napomenimo da je izbor konkretne realizacije tog grafa, za koji smo dosad dali samo inspiraciju, subjektivan. Implicitna shema grafa dana je u 4.1, no u ovom projektu uvijek nastojimo omogućiti jednostavno stvaranje varijanti na glavnu izvedbu. Doista, za što je moguće temeljitije izvlačenje informacija iz koda, potrebno je obratiti pažnju na veliki broj specijalnih slučajeva. Specijalnih slučajeva ne samo da je puno, zbog čega će vjerojatno uvijek biti novih ideja ili mišljenja, već njihovim uvođenjem povećavamo kod i kompliciramo njegovu strukturu. Zato se u ovom trenutku fokusiramo na osnovne funkcionalnosti sa što elegantnijom strukturom.

Spomenutu transformaciju obavljamo u Pythonu, ali primijetimo da je neke ideje moguće izvesti i unutar same baze. Kad je to moguće, dobro je tako napraviti zbog boljih performansi i deklarativnog koda.

3.1 Obrada jednog modula

Započnimo s problemom analize jednog modula[1], odnosno preslikavanja njegovog koda u graf kakav želimo. U tom problemu imamo dva glavna cilja:

- *Obilazak AST-a i izvlačenje informacija o imenima.* Prilikom obilaska AST-a, želimo trenutnom vrhu dodati atribute (npr. tip ako je dan) i povući odgovarajuće bridove s nekim drugim vrhovima u grafu. U preorder obilasku stabla, obilazimo sintaktičke elemente onim redom kojim su navedeni u kodu. Osim toga, posebice za povlačenje bridova, potrebno nam je nekakvo praćenje konteksta, budući da u čistom obilasku u svakom trenutku vidimo samo jedan vrh. U našoj implementaciji koristimo se strukturom `SNode` koja osim podataka koje prenosimo u bazu, sprema i neke podatke važne za kontekst. Pri obilasku koristimo razne `handler`-e, svaki od kojih je povezan sa specifičnom vrstom ili vrstama vrha u AST-u. Na taj način enkapsuliramo željene funkcionalnosti unutar jedne funkcije i dozvoljavamo fleksibilnost po pitanju njihovog izbora. Moguće je pri obradi jednog vrha dodati njegovu djecu na stog za kasniju obradu, ali ih i ignorirati ili obratiti odmah ako je to moguče ili prikladno.

- *Rezolucija imena.* Programski kod sastoji se od raznih imena, a njihovo značenje ovisi ne samo o dinamičnom pridruživanju već i o nazivnom prostoru u kojemu se nalaze. Ključno je razlikovati dva imena koja se nalaze u različitim prostorima (npr. svaka od dvije funkcije može imati vlastitu lokalnu varijablu `x`), ali i prepoznati dva imena kao ista (posebice prilikom uvoza).

Ta dva cilja ispunjavamo pomoću tri obilaska:

1. *Prvi obilazak.* U prvom obilasku prolazimo kroz tablicu simbola. Za svaki novi lokalni simbol dodajemo novi `SNode` i postavljamo kontekstualne veze — roditelja i rječnik s vlastitom djecom kad simbol ima vlastiti nazivni prostor.

2. *Drugi obilazak.* U drugom obilasku obilazimo AST i sve atribute dosad viđenih simbola dodajemo kao nove simbole.

[1] u ovom kontekstu, izraz *modul* poistovjećujemo s `.py` datotekom

3. *Treći prolazak.* U trećem obilasku ponovo obilazimo AST. Fokusiramo se na ranije spomenute funkcionalnosti važne kod obilaska i uglavnom stvaramo bridove.

3.2 Proširenje na cijeli paket

Da bismo analizirali cijeli paket[2] koji se sastoji od većeg broja modula i potpaketa, potrebno je doraditi dosadašnji algoritam:

- Najprije je potreban još jedan "nulti" obilazak, ovaj put fileova u repozitoriju, kojim nalazimo sve module i pakete (`__init__.py` može definirati uvoz i varijable na razini paketa) koje moramo analizirati. I ovdje je bitno uspostaviti kontekstualne veze između paketa i modula.
- Izvršavamo prvi obilazak svakog od modula[3] u proizvoljnom poretku.
- U drugim obilascima, koje opet izvršavamo u proizvoljnom poretka, osim atributa analiziramo i uvoz (*import*) dodajući u graf relevantne bridove. Ovo je bitno da bismo u trećem prolasku mogli uvezena imena ispravno povezati s vrhovima. Naime, ako modul `X` uvozi iz `Y`, a `Y` iz `Z`, tada će prvo biti potrebno uvesti simbole iz `Z` u `Y`, budući da `X` tranzitivno uvozi i iz `Z`.
- Treći obilazak izvršavamo kao ranije, ali u specifičnom poretku — nikad ne počinjemo treći obilazak modula prije nego obiđemo one module iz kojih uvozi. To je moguće pod pretpostavkom da nema cirkularnosti u uvozu, što, iako nije samo po sebi zabranjeno, može dovesti do nepredividivog ponašanja.

Uvoz, uvoz iz vanjskih paketa ili modula, dakle onih koji se očekuju naći u okruženju ali nisu dio samog paketa, je u posebnoj kategoriji. Iako je moguće u graf dodati i takva imena, primjerice kao elemente dodatnog virtualnog potpaketa, to bi sa sobom vuklo neke komplikacije koje su jasne iz 6.1. Zbog toga takva imena ignoriramo.

[2] u ovom kontekstu, *paket* poistovjećujemo s mapom (folderom); u prošlosti je mapa morala imati `__init__.py` file da se smatra paketom, no u novijim verzijama ($\geq$3.3) nije obavezan

[3] ovdje mislimo i na module i na pakete s `__init__.py`-jem

4 Baza

4.1 Implicitna shema

U ovom potpoglavlju dajemo konkretnu implicitnu shemu naše grafovske baze, uz ponovnu napomenu kako je uz modifikacije moguće postići razne varijacije na temu, ovisno o osobnim izborima ili specijaliziranoj namjeni.

Sljedeće su prisutni tipovi (labele) vrhova: `Package`, `Module`, `Class`, `Function` i `Name`. Glavno svojstvo svih vrhova je `fullname` koje predstavlja puno ime objekta iz perspektive korijena paketa. Ono se razlikuje između svaka dva različita vrha. Druga zajednička svojstva su `name`, `moduleName` i `packageName` koji su imena bez prefiksa i olakšavaju neke vrste upita. Daljnja svojstva ovise o tipu — moduli, klase i funkcije mogu imati `docstring` (dokumentacijski string), funkcije imaju svojstvo `isAsync` koje govori je li funkcija asinkrona, a ostala imena mogu imati zabilježen tip (npr. `int` ili `str`).

Moguće je bilo dodati još tipova (npr. koji predstavljaju posebno metode, atribute ili argumente) no to semantički prikazujemo bridovima na način koji ćemo opisati u nastavku.

Sada navodimo tipove bridova:

- `WIHIN_SCOPE`. Govori da je prvi vrh definiran u nazivnom prostoru drugog. Implicitno, prvi vrh je ime, klasa ili funkcija, dok je drugi paket, modul, klasa ili funkcija. Nadalje nećemo uvijek specificirati moguće tipove vrhova.
- `ASSIGNED_TO_WITHIN`. Govori da je prvom vrhu pridružena vrijednost unutar opsega drugog.
- `REFERENCED_WITHIN`. Govori da je prvi vrh "spomenut" unutar opsega drugog.
- `IMPORTED_TO`. Govori da je prvi vrh uvezen u opseg drugog.
- `IMPORTS_FROM`. Suprotno, drugi vrh uvezen je u opseg prvog. Ovaj brid nije uvijek dualan prethodnom jer se odnosi isključivo na module i pakete.
- `INHERITS_FROM`. Jedna klasa naslijeđuje drugu.
- `METHOD`. Označava funkciju metodom klase.
- `DECORATES`. Označava da je prvi vrh dekorator drugom.
- `ARGUMENT`. Označava da je vrh argument u funkciji.
- `ATTRIBUTE`. Označava da je vrh atributut drugog.
- `RETURNS`. Govori da se vrh "spominje" u `return` naredbi funkcije. Ako postoji logičko grananje ili drugi oblik kontrole toka programa, to se ignorira. Ovaj brid ne govori što točno funkcija vraća, već samo ističe da postoji veza između vrha i vraćenog.
- `ASSIGNED_TO`. Govori da se prvi vrh "spominje" u definiciji drugog. Ponovo, kontrola toka ili višestruka pridruživanja se ignoriraju. Brid samo ističe da postoji veza, a ne pokušava otpetljati kontrolu toka, dinamičko stanje ili smisao.
- `TYPED_WITH`. Označava da je vrh tipiziran klasom. Ako nije tipiziran klasom definiranom unutar paketa nego *built-in* tipom, taj podatak se sprema kao svojstvo vrha.

U trenutnoj izvedbi bridovi najčešće nemaju svojstva, a iznimka je `alias` kod uvoza.

4.2 Aplikacija

Funkcionalnosti paketa dostupne su preko CLI sučelja. Osnovnoj naredbi `pygdb` možemo dodati argumente kojima određujemo URI servera baze, podatke za autentifikaciju i ime baze. Ukoliko nisu dani, koriste se vrijednosti zadane unutar koda.

Nakon toga, dostupne su tri podnaredbe (*subcommands*). Među njima je ključna `add` kojom analiziramo paket i dobiveni graf šaljemo u bazu. Detaljnije o tom problemu raspravljamo u 4.3. Paket je dan kao URI koji može biti lokacija na lokalnom računalu ili druga mapa dohvatljiva naredbom `git clone`. Drugi argument je stupanj *logging*-a.

Nakon što se paket prenese u bazu, upite je moguće vršiti unutar Neo4j Browser sučelja, ali i pomoću podnaredbe `query`. Rezultati upita vizualiziraju se pomoću Graphviz-a. Ta opcija je potencijalno vrlo spora i nesigurna, dakle treba ju koristiti s oprezom te eventualno ne izložiti korisnicima s nedovoljnim ovlastima ili za iste dodati funkcionalnosti koje bi onemogućile Cypher injection.

Treća podnaredba `clear` (alternativno `create`, `reset` ili `start`) dozvoljava da se baza resetira, u smislu da se izbriše i stvori nova. Nova baza će se također stvoriti ako ne postoji, a i dodat će se odabrana ograničenja i indeksi. Dodajemo ograničenje jedinstvenosti svojstva `fullname`[4], ograničenje postojanja svojstva `name` kao i *fulltext* indeks na istom budući da očekujemo da će upravo to svojstvo često sudjelovati u upitima.

Za kraj potpoglavlja dajemo jedan primjer potpune naredbe:

```
python3 pygdb --server bolt://localhost:7689/ \
--auth neo4j password --database pygdb \
add --uri git@github.com:numpy/numpy --name numpy
```

4.3 Prijenos podataka i performanse

Najjednostavniji način za prijenos podataka iz internih objekata u bazu je stvaranjem zasebne transakcije za svaki vrh ili brid. Ta metoda ima dodatnu prednost jednostavnog i fleksibilnog koda, u smislu da istu funkciju možemo koristiti za sve tipove vrhova ili bridova i za bilo kakvu konfiguraciju svojstava. Drugim riječima, u ovoj metodi možemo unositi promjene u postojeće tipove i svojstva bez da moramo imalo mijenjati kod za prijenos u bazu. Velika mana ove metode je da je vrlo spora i to naročito u stvaranju bridova — njih je više i njihovo stvaranje zahtijeva dva `MATCH`-a.

Alternativa je uporaba *batch*-upita kako je preporučeno i pokazano u [7]. Ova metoda doista znatno skraćuje potrebno vrijeme i čini ga podnošljivim i za veće pakete. Mana je u tome da, kao što je objašnjeno u [6] parametri ne mogu odgovarati labelama vrha ili tipu brida, nazivu (ključu) svojstva niti su podržane parametarske mape kao u prošlom slučaju. Zbog toga je tipove i atribute potrebno hardkodirati, što komplicira stvaranje izmjena u implicitnoj shemi.

Druga mogućnost za ostvarivanje boljih performansi je asinkroni driver. Naravno, veću brzinu ne očekujemo u slučaju mnoštva jednostavnih transakcija, već ga koristimo samo u *batch* varijanti. Taj driver je po navodu službene dokumentacije eksperimentalan, stoga postoji i sinkrona verzija aplikacije u `sync_main.py`.

Razliku u performansama možemo prikazati na primjeru paketa PyMC (kasnije u 5.1) s otprilike 66 tisuća bridova. Dok kraj u prvoj varijanti nije bio ni na vidiku, u drugoj je cjelokupni prijenos trajao 20 minuta. Uvođenjem asikronog drivera, to se vrijeme gotovo pa prepolovilo.

[4] vrhovi se upravo drže u rječniku u kojemu su `fullname` ključevi, ali ovo ograničenje je svejedno korisno da spriječi korisnika da zabunom unese isti paket dvaput; time se na `fullname` dodaje i indeks

5 Primjeri

5.1 Pokazna baza

5.2 Primjeri upita

5.3 Rezultati

6 Osvrt

6.1 Nedostaci

Pythonova dinamičnost i fleksibilnost donose ozbiljne nedostatke ovakvom projektu od početka. Navedimo tri značajna uzroka:

1. *Fleksibilnost u pridruživanju imena.* Kao što smo već spomenuli u 1.2, isto ime u toku programa može biti vezano za više potpuno različitih objekata. Zbog toga, naši rezultati ne moraju u svim slučajevima biti pouzdani, a mogu ispasti i zbunjujući. Ipak, takvo recikliranje imena nije dobra praksa i ne očekujemo da ćemo ga često viđati.

2. *Dinamički generirani kod.* U Pythonu su mogući razni oblici koda koje možemo smatrati dinamički generiranim. Takav kod je onda teško, ako uopće moguće, analizirati unutar apstraktnog sintaksnog stabla. Jedan primjer je Pythonova funkcija `exec` koja dan joj string izvodi kao kod. Korištenje te funkcije je rijetkost i najčešće nije dobra ideja.

   Ipak, postoje drugi primjeri koji su puno češći. Primjer su funkcije `getattr` i `setattr` koje upravljaju s atributima objekta. Pritom se atribut daje kao string, a ne mora biti `ast.Name` vrh već može biti rezultat evaluacije nekog izraza ili iteratora.

   Drugi važni primjer su funkcije iz standardne biblioteke `importlib` koja omogućava veću kontrolu nad uvozom modula i paketa. Ponovo, značenje njihovih argumenata može se skrivati iza evaluacije kompliciranog i dinamički određenog izraza. Uporaba biblioteke `importlib` upravo je češća u sofisticiranijim paketima koji i zahtijevaju takve dodatne funkcionalnosti. To predstavlja značajan problem unutar ovog projekta jer ne možemo povezati imena uvezena na taj način s objektima. Preostaje izbor — stvarati duplikate za vrhove ili ignorirati sva imena koja ne možemo rezolvirati. Duplicirani vrhovi imaju memorijsku cijenu, a sama njihova prisutnost ne garantira uvijek da će biti kao takvi prepoznati.

3. *Objekti uvezeni iz koda u drugom jeziku.* U Pythonu je moguće uvesti objekte i imena definirane u drugim programskim jezicima, a poznat su i česti slučaj C i C++. Tada ni ta imena ne možemo rezolvirati bez značajnog dodatnog napora. Znamenit primjer su Pythonovi *built-in* moduli. Primjerice, `ast.py`, koji je zapravo omotač, uvozi modul `_ast.py` kreiran iz izvornog koda u C-u (v. [3]).

6.2 Slični projekti

U trenutku pisanja rada, autor nije bio svjestan projekata koji bi odgovarali ovom u funkcionalnostima, naime izvlačenju semantičkih pojava u kodu u Pythonu, njihovom modeliranju grafom i spremanju u grafovsku bazu podataka. Navedimo dva projekta koji su donekle slični:

- `pyclrb.py` (v. [8]). Ovaj modul Pythonove standardne biblioteke analizira klase i funkcije unutar jednog modula. Za to se također koristi obilaskom AST-a.

- `Pyan` (v. [4]). Ovaj paket, vjerojatno najsličniji našem, statički analizira kod u jednoj ili više datoteka i konstruira graf koji se može vizualizirati pomoću Graphviza. Također se koristi AST-ovima i tablicama simbola. Osim razlika u implementaciji, glavna je razlika da je konkretno riječ o statičkom *call graph*-u, dakle paket se bavi samo funkcijama, metodama, i njihovim međusobnim pozivanjem.

Pored ovih paketa, postoji još obilje drugih koji s vlastitim, specifičnim, namjenama vrše statičku analizu koda u Pythonu, pritom se najčešće služeći sličnim alatima.

Literatura

[1] Luka Šimek (lsimek2). *nbp-projekt*. 2024. URL: https://github.com/lsimek2/nbp-projekt.

[2] *ast – Abstract Syntax Trees*. Python Software Foundation. URL: https://docs.python.org/3/library/ast.html (pogledano 14. 5. 2024.).

[3] Python Software Foundation. *Python-ast.c*. 2024. URL: https://github.com/python/cpython/blob/main/Python/Python-ast.c (pogledano 6. 6. 2024.).

[4] David Fraser. *pyan*. 2011-2021. URL: https://github.com/davidfraser/pyan/ (pogledano 6. 6. 2024.).

[5] Thomas Kluyver. *Green Tree Snakes - the missing Python AST docs*. 2012. URL: https://greentreesnakes.readthedocs.io (pogledano 19. 5. 2024.).

[6] Neo4j. *Parameters*. URL: https://neo4j.com/docs/cypher-manual/current/syntax/parameters/ (pogledano 6. 8. 2024.).

[7] Neo4j. *Performance recommendations*. URL: https://neo4j.com/docs/python-manual/current/performance/ (pogledano 6. 8. 2024.).

[8] *pyclbr — Python module browser support*. Python Software Foundation, 2024. URL: https://docs.python.org/3/library/pyclbr.html (pogledano 6. 6. 2024.).

[9] *symtable — Access to the compiler's symbol tables*. Python Software Foundation. URL: https://docs.python.org/3/library/symtable.html (pogledano 25. 5. 2024.).