

TDK Dolgozat

Rózsa Balázs

Python kód automatikus átalakítása strukturált mintaillesztés utasítás támogatására

Készítette:

Rózsa Balázs

programtervező informatikus

3. évf. BSc hallgató

Témavezetők:

Antal Gábor

tudományos segédmunkatárs

Dr. Ferenc Rudolf

egyetemi docens

Szegedi Tudományegyetem Informatikai Intézet

Szoftverfejlesztés Tanszék

Szeged

2022

Tartalomjegyzék

1. Bevezetés	3
2. Kapcsolódó munkák	4
2.1. Verziók közötti kompatibilitást elősegítő refaktorálás	5
2.2. Optimalizálást elvégző refaktorálás	5
2.3. Biztonságot növelő refaktorálás	6
2.4. Kódtranszformáló keretrendszerek	6
3. Szerkezeti mintaillesztés	7
3.1. Szerkezeti mintaillesztés a Python programozási nyelvben	8
3.1.1. Áttekintés	9
3.1.2. Kulcsszavak, definíciók	10
3.2. Minták	11
3.2.1. Class Pattern	12
3.2.2. Literal Pattern	13
3.2.3. Or Pattern	13
3.2.4. Hozzárendelő minták	13
4. AST	15
4.1. Python AST	16
4.2. Vizsgált elemek	17
4.2.1. If node	17
4.2.2. BoolOp node	17
4.2.3. Compare node	18
4.2.4. Match node	19

4.2.5. Match[Pattern] node-ok	19
5. Transzformációs rendszer	21
5.1. Forráskód transzformáció	22
5.2. Transformer	23
5.3. Analyzer	25
5.4. Pluginok	26
6. Megvalósított átalakítások	28
6.1. Felismerhető minták	30
6.1.1. LiteralPattern	30
6.1.2. SingletonPattern	30
6.1.3. OrPattern	31
6.1.4. GuardPattern	32
6.1.5. ClassPattern	33
6.2. Egyésba ágyazott feltételes elágazások	35
6.2.1. Rekurzív transzformálás	36
6.2.2. "Lapítás"	36
7. Tesztelés és kiértékelés	39
7.1. Statisztikák	40
7.2. Talált hibák	41
8. Összegzés	44
Irodalomjegyzék	46

1. fejezet

Bevezetés

A Python 3.10-es verziójában bevezették a szerkezeti mintaillesztést, mellyel a komplex adatok könnyebb, olvashatóbb, és karbantarthatóbb kezelését segítik elő. Segítségével képesek lehetünk az adat struktúrája alapján elágaztatni a programkódot, továbbá az adatot lehetőségünk van további részekre bontani a későbbi feldolgozás érdekében. Ez a fajta programvezérlés szerkezeti mintaillesztés nélkül csak feltételes elágazások használatával lehetséges, melyeknek az egymásba ágyazásával, és egymás után láncolásával tudjuk elérni a szerkezeti mintaillesztéssel ekvivalens működést. A szerkezeti mintaillesztés szabályai, és működése a 3. fejezetben részletesen be van mutatva.

A dolgozatom célja egy olyan program implementálása, amely képes a feltételes elágazásokat lehetőség szerint átalakítani az új, szerkezeti mintaillesztésre, oly módon, hogy a program működése ne változzon, továbbá transzformáció után karbantarthatóbb, olvashatóbb legyen az átalakított forráskód. A megvalósított program áttekintése az 5. fejezetben található, és transzformáció pontos szabályai és korlátai pedig a 6. fejezetben kerülnek bemutatásra.

Automatikus kódátalakítás során a legtöbbször felhasznált eszköz az absztrakt szintaxis fa, azaz AST. Az AST a programozási nyelv nyelvtanát reprezentáló adatszerkezet, melyet a legtöbb esetben a fordítóprogramok generálnak. Az AST eredetéről, felépítéséről, és a program során vizsgált elemekről a 4. fejezetben lesz szó.

2. fejezet

Kapcsolódó munkák

Az automatikus forráskód átalakítás problémája nem újkeletű, viszont még napjainkban is rendkívül népszerű kutatási és fejlesztési területnek tekinthető. Kódtranszformáció során elsődleges szempont, hogy a kód viselkedése ne változzon, viszont a kód struktúrája valamilyen szempontból javuljon. Az ilyen fajta kódátalakítást **refaktorálásnak** nevezzük, mely kifejezést elsőként *Martin Fowler* definiálta az 1999-ben kiadott *Refactoring* című könyvében [11].

A refaktorálásnak számos célja lehet [18]: olvashatóság javítása, biztonság növelése, programozási nyelvek (verziói) közötti fordítás, optimalizálás, stb. Azonban céltól függetlenül, amennyiben nem manuálisan kívánjuk elvégezni ezeket a módosításokat, (melyek sok esetben rendkívül időigényesek lehetnek) akkor szükségünk lesz a forráskód valamilyen módon történő strukturális reprezentációjára.

A forráskódot legtöbbször egy fa alapú adatszerkezetben szokták ábrázolni, mely könnyebben kezelhető, és több információt tartalmaz, mint a szöveges formában lévő forráskód, illetve a legtöbb fordítóprogram is egy fa struktúrát épít a forráskódból saját felhasználásra. Ezt az adatszerkezetet általában egy statikus kódelemzést végző rendszer szokta előállítani, mely rendszerekről számos publikáció megjelent [3], [8].

Léteznek specifikus, meghatározott célokra fejlesztett forráskód átalakító rendszerek, azonban számos keretrendszer is készült, melyek szabadon konfigurálhatóak, tetszőleges átalakításokat végeztethetünk el velük. A következőkben bemutatok néhány forráskód átalakító rendszert.

2.1. Verziók közötti kompatibilitást elősegítő refaktorálás

Gyakori probléma a rendszeresen frissülő programozási nyelveknél, hogy a fejlesztők szeretnék kihasználni az új verziók által biztosított lehetőségeket, azonban a verziók közötti konverzió nem triviális.

A Python 3.0 verziója már nem volt visszamenőlegesen kompatibilis a Python előbbi verzióival, ezért létrehoztak egy olyan eszközt, mely képes a 2.x-es verzióban írt forráskódot automatikusan 3.x-el kompatibilis forráskóddá alakítani. Bár a transzformáció nem végezhető el teljesen automatikusan, a *2to3* [21] program segítségével a folyamat nagy része automatizálható.

Örökölt (*Legacy*) rendszereknél előjöhethet az igény arra, hogy az újabb verzióban írt programot "fordítsuk vissza" régebbi verziókra, hiszen nincs lehetőségünk az örökölt rendszert frissíteni, azonban szeretnénk kihasználni az új verzióban bevezetett funkciókat. Antal Gábor és társai [4] létrehoztak egy olyan automatikus refaktoráló eszközt, mely képes C++11-ben írt forráskódot automatikusan C++03 verzióval kompatibilis forráskóddá alakítani, ezzel elősegítve a fejlesztők munkáját.

2.2. Optimalizálást elvégző refaktorálás

Sokszor szembesülünk a ténnyel, hogy az emberek nem írnak olyan optimális kódot, mint amilyet szeretnének. Ez természetes, hiszen a forráskód az emberekre van "optimalizálva". A fordítóprogramok fordítás során rengeteg optimalizáló átalakítást [19] végeznek el, hogy a program futása a lehető leghatékonyabb legyen. Továbbá minimalizálható egy program futásának az *energiafogyasztása* [6], mely tényező különösen fontos a beágyazott rendszerek – főleg a mobileszközökre szánt rendszerek – tervezése során.

Magát a fejlesztési folyamatot is "optimalizálhatjuk": finomíthatjuk a memóriában tárolt *kollekciók lekérdezéseit* [12], vagy új, erősebb eszközökkel *bővíthetjük a fejlesztők matematikai eszköztárát* [26].

2.3. Biztonságot növelő refaktorálás

Amennyiben nem nyílt forráskódú a szoftverünk, a legtöbb esetben elkerülendő, hogy a felhasználók hozzájussanak a forráskódunkhoz. Azonban egy (elegendő idővel, és tapasztalattal rendelkező) fejlesztő bármilyen gépi kódot képes visszafordítani forráskóddá.

Ennek a feladatnak a megnehezítésére jó megoldás lehet a *forráskód obfuszkáció*¹, [9], mely tovább nehezítheti a forráskód visszafejtés (*reverse engineering*)² folyamatát. További védelmet jelenthet a *forráskód randomizáció* [17], mely hasonló a memória címterület randomizációjához³.

Plágium elleni védelemként használhatjuk az AST alapú kódösszehasonlítást [30], illetve akár a FaCoY (Find a Code other than Yours) [14] forráskód-kereső rendszert.

2.4. Kódtranszformáló keretrendszerek

A legtöbb fejlesztői környezet tartalmaz valamilyen refaktoráló keretrendszert: PyCharm⁴, Eclipse⁵, IntelliJ⁶, stb. Segítségükkel ellenőrizhetjük a kódunk helyességét, fejlesztési irányelveket érvényesíthetünk, és optimalizálhatjuk a kódunkat.

Azonban vannak, speciálisan a refaktorálásra elkészített keretrendszerek, melyeknek pontosan definiálhatjuk hogy mit, és hogyan alakítsanak át. Ilyen például a *ROSE* keretrendszer [29], mely egy nyílt forráskódú, kód analízálásra, és transzformálásra használható fordítóprogram alapú rendszer, mely képes C, C++, és Fortran kódok feldolgozására.

Hasonló keretrendszer a *Proteus* [28], mely egy C / C++ kód refaktorálásra készített rendszer. Sajátossága, hogy különös figyelmet fordít a forráskód stílusának a megőrzésére. További példaként felhozható még a *Spoon* keretrendszer [20], mely a Java nyelvhez készült, kód analízálásra és transzformálásra felhasználható keretrendszer.

¹[https://en.wikipedia.org/wiki/Obfuscation_\(software\)](https://en.wikipedia.org/wiki/Obfuscation_(software))

²https://en.wikipedia.org/wiki/Reverse_engineering#Binary_software

³https://en.wikipedia.org/wiki/Address_space_layout_randomization

⁴<https://www.jetbrains.com/pycharm/>

⁵<https://www.eclipse.org/>

⁶<https://www.jetbrains.com/idea/>

3. fejezet

Szerkezeti mintaillesztés

Ahogy a felhasznált adatok egyre komplexebbek lettek, úgy a programozási nyelvek is egyre több módot biztosítottak a fejlődésük során ezek kezelésére. Absztrakt adattípusok, osztályok, és objektumok használatával nem csak az adatok tárolására van lehetőség, a programozási nyelvek ezen adatok manipulációjára is biztosít módot.

Egy ilyen mód a szerkezeti mintaillesztés, mely két feladatot lát el: a komplex adatokat képes releváns részekre bontani a későbbi feldolgozás érdekében. Továbbá, a feltételes elágazással szemben - ami az adat konkrét értékétől függően készít elágazásokat - a szerkezeti mintaillesztés az adat struktúrájától függően is képes elágazni a programkodot. Ez a fajta adatkezelés először a funkcionális programozási nyelvekben (pl. Haskell: lásd 1. kódrészlet) jelent meg, azonban idővel objektum orientált nyelvekben [1] [2] is bevezették.

Szerkezeti mintaillesztés során a két fő tényező az adat (az *Alany*), és a minták. A mintaillesztés azon a feltételezésen alapszik, hogy az *Alany* egy bizonyos szerkezeti mintát követ, mely, ha bebizonyosodik, társítható hozzá egy, az adat szerkezeti mintájára illő, specifikus adatfeldolgozásért felelős metódus. Kondicionális mintaillesztésről beszélünk, ha ezen feltételezés mellé még egy plusz, a mintától független feltétel is társul a mintaillesztéshez. Számos minta és a hozzá tartozó adatfeldolgozási műveletek kombinációjából képesek vagyunk egy olyan rendszert létrehozni, amely biztosítani tudja az adatok kezelését az adat struktúrájától függően, akár kondicionális, akár un kondicionális módon. [16] A mintaillesztési folyamat elsődleges "kimenete" a mintaillesztés sikeressége, azonban tekinthető másodlagos "kimenetnek" a mintára illeszkedő értékek változókhöz rendelése.

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Kódrészlet 1: Szerkezeti mintaillesztés a Haskell programozási nyelvben

Ebben a fejezetben összefoglaló jelleggel bemutatom a Python 3.10-es verziójában bevezetett ¹ szerkezeti mintaillesztés használatát, működését, és szabályait melyek a későbbiek során rendkívül fontosak lesznek a transzformációs logika megértéséhez. Amennyiben az olvasó részletesebben szeretne elmélyülni a szerkezeti mintaillesztés témakörében, erősen ajánlom a Brandt Bucher és Guido van Rossum által írt PEP635 [15], és PEP634 [7] -et, illetve a funkcióért felelős fejlesztők által írt tudományos cikket [16], melyeken a fejezet alapszik.

3.1. Szerkezeti mintaillesztés a Python programozási nyelvben

Az utóbbi években az adatelemzés és – feldolgozás területén a dinamikus nyelvek egyre népszerűbbeké váltak. A dinamikus nyelvek viszonylag könnyű megközelíthetősége, kombinálva a nyelvekhez tartozó rendkívül optimalizált adatelemző és -feldolgozó könyvtárakkal elősegítették ezeknek a nyelveknek a feltörését [25].

A Python nyelv különösen releváns lett ezen a területen. Könnyen tanulható, és számtalan könyvtárral rendelkezik mind az adatfeldolgozás és – elemzés, mind pedig a mesterséges intelligencia ² területén. A Pythonban megtalálható széleskörű adatkezelési lehetőségek ellenére, a 3.10-es verzió előtt a szerkezeti mintaillesztés csak limitált formában volt elérhető.

Szekvenciális adatokból lehetőségünk van individuális adatokat kinyerni, és változókhoz rendelni őket ("Iterable unpacking" segítségével)³. Azonban nem minden esetben érdemes az adatokat szekvenciákban tárolni, sokszor kézenfekvőbb megold-


¹<https://docs.python.org/3/whatsnew/3.10.html>

²<https://gist.github.com/sebp/58da862b779489998e8e6088908fbfa5>

³<https://www.pylenin.com/blogs/unpacking-iterables-in-python/>

dás lehet gráf struktúrák, illetve osztályok és objektumok használata az adatok reprezentálására. Az ilyen objektumoknak a "feltérképezésére" - típusának és szerkezetének a megállapítására, és ez alapján elágazások létrehozására szükségszerűen az `if ... elif ... elif ... else` alakú feltételes elágazásláncokat kellett használnunk, `isinstance(x, cls)`, `hasattr(x, cls)`, stb. típusú ellenőrző függvényeket használva.

Egy elegánsabb [27], olvashatóbb megoldást ígérve vezették be a teljes szerkezeti mintaillesztést a Python 3.10-es verziójában. A szintakszisa hasonló a más nyelvekben megtalálható *switch/case* szerkezetekhez, azonban ezeknél komplexebb mintákat is támogat, ahogy az a 2. kódrészletben is látható.



```
if (    key == 37 or
        key == "ArrowLeft"):
    return "←"
elif ( key == 38 or
        key == "ArrowUp"):
    return "↑"
elif ( key == 39 or
        key == "ArrowRight"):
    return "→"
elif ( key == 40 or
        key == "ArrowDown"):
    return "↓"
else:
    return "?!"
```

```
match key:
    case 37 | "ArrowLeft":
        return "←"
    case 38 | "ArrowUp":
        return "↑"
    case 39 | "ArrowRight":
        return "→"
    case 40 | "ArrowDown":
        return "↓"
    case _:
        return "?!"
```

Kódrészlet 2: Szemmel láthatóan elegánsabb a szerkezeti mintaillesztés használata.

3.1.1. Áttekintés

A mintaillesztési folyamat "bemenetként" egy mintát vár, és egy értéket (az *Alanyt*). A folyamatot talán a legegyszerűbben így lehetne megfogalmazni: "Az *Alany* értékét a mintára illesztjük". A mintaillesztés sikerességét tekintjük a folyamat elsődleges "kimenetének".

Bizonyos minták azonban képesek (sikeres illesztés esetén) az *Alanyt* vagy annak valamely részét egy új, lokális változóhoz rendelni (*Binding*), melyet felhasználhatunk

a későbbiek során (akár a teljes mintaillesztési blokk után is). A *Binding*-ot tekinthetjük a mintaillesztési folyamat másodlagos "kimeneteként". Fontos azonban, hogy sikertelen mintaillesztés esetén nem definiált, hogy mi történik az összerendelt változókkal. Ezt szándékosan hagyták definiálatlanul, hogy elkerüljék a szemantikus korlátokat, mellyel limitálhatnák a *Binding* későbbi bővítését [7].

Sok esetben egy minta tartalmazhat egy vagy több almintát. Ebben az esetben a mintaillesztés sikeressége az alminták illesztésének a sikerességén múlik. Hasonlóan a logikai kifejezések kiértékelési sorrendjéhez ⁴, az alminták is balról jobbra értékelődnek ki, egészen addig, amíg a mintaillesztés sikeressége nem lesz egyértelmű.

Minden elágazáshoz adhatunk egy plusz feltételt, úgynevezett *Guard*-ot. A *Guard* tulajdonképpen csak egy logikai kifejezés, melynek értéke csak sikeres mintaillesztés esetén kerül kiértékelésre. Segítségével további feltételeket szabhatunk ki a mintákra. Fontos, hogy míg a minta csak az *Alanyt* vizsgálhatja, a *Guard*-ra nincsen ilyen megszorítás.

3.1.2. Kulcsszavak, definíciók

A **match** kulcsszó jelzi a mintaillesztési blokk kezdetét. A kulcsszó után kell megadni az *Alanyt*, melynek az értéke lesz felhasználva a mintaillesztés során. Amennyiben több értéket, vesszővel elválasztva adunk meg, egy **tuple** objektum fog létrejönni az értékekből. Ez után, a szokásos Python indentálási szabályoknak megfelelően [27] következnek az elágazások.

A **case** kulcsszóval definiálunk egy új elágazást. A kulcsszó után kell megadnunk a mintát, amelyre az *Alany* értéke lesz illesztve. Továbbá, a minta után adható meg, **if** kulcsszó után, az elágazás *Guard*-ja, melynek logikai értéke csak sikeres mintaillesztés esetén fog kiértékelődni. Ezt követően, új sorban, indentálva következik a kódszegmens, mely sikeres mintaillesztés esetén fog végrehajtódni.

Vannak olyan minták melyek önmagukban "**tagadhatatlanok**", azaz nyelvtanilag bebizonyítható, hogy bármilyen *Alany* értékre sikeresen illeszkednek. Ezekkel megadható, mintaillesztési blokkonként egy, alapértelmezett ág (lásd 3. kódrészlet).

Az elágazások kódsorrendben, fentről lefelé értékelődnek ki. A mintaillesztési folyamat során a legelső olyan elágazás fog végrehajtódni, ahol a mintaillesztés sikeresnek

⁴<https://docs.python.org/3/reference/expressions.html>

bizonyult (És a *Guard* értéke is igazra értékelődött ki, amennyiben volt). Emiatt a sorrendiség miatt csakis a legutolsó elágazás lehet "tagadhatatlan", hiszen bármely utána következő ág garantáltan sosem futna le. Ezért a Python értelmező hibát dob olyan esetben, ha egy "tagadhatatlan" ág után további elágazásokat definiálunk.

3.2. Minták

A mintáknak két fő szerepük van a mintaillesztés során: (szerkezeti) korlátokat, feltételeket szabnak az *Alanyra*, és definiálják, hogy az *Alany* mely értékei legyenek hozzárendelve új változókhoz. A már említett "Iterable unpacking"-ben, amit tekinthetünk a szerkezeti mintaillesztés elődjének Pythonban, egyetlen egy szerkezeti minta létezik (szekvenciák reprezentálására), azonban számos hozzárendelő minta adott. A teljes szerkezeti mintaillesztés ezzel szemben megannyi szerkezeti mintát definiál, viszont minimális számú hozzárendelő mintát.

Az "Iterable unpacking"-gel ellentétben sikertelen mintaillesztés esetén nem keletkezik hiba. Emiatt törekedtek arra, hogy a mintáknak a mellékhatásai minimálisak legyenek. Hiszen a mintáknak az egyik legfontosabb funkciójuk az egymásba ágyazhatóság. Képzeljük el azt a szituációt, ahol egy al minta mintaillesztése során valamilyen változóhoz egy érték kerül hozzárendelésre, azonban ezután a szülő minta mintaillesztése sikertelennek bizonyul. Ebben az esetben nehéz lenne a már hozzárendelt változókat visszaállítani. Ezért nem megengedett, hogy a minták objektumok attribútumaihoz, vagy indexelt értékekhez rendeljenek hozzá értéket. Így minimalizálják a változó hozzárendelés esetleges mellékhatásait, hiszen így csak új, lokális változókhoz lehet értéket hozzárendelni.

Számos minta első ránézésre tekinthető kifejezésnek. Fontos azonban megjegyezni, hogy ez semmiképpen sincs így, a minták nem (tartalmaznak) kifejezések(et). A legcélravezetőbb módszer a minták megértésére talán az, hogy tekintsünk rájuk deklaratív elemekként, melyek hasonlóak a függvénydefinícióknál megszokott formális paraméterekhez ⁵ [15].

A dolgozatomban nem célom bemutatni az összes definiált mintát, azonban a következőkben bemutatom a transzformációs folyamat megértéséhez elengedhetetlen mintákat.

⁵<https://stackoverflow.com/a/45065422>

3.2.1. Class Pattern

A class pattern-nek két fő feladata van: Megállapítani, hogy az *Alany* tényleg a mintában definiált osztály egy objektuma-e, illetve az *Alany* különböző attribútumaira is képes mintákat illeszteni. Az *Alany* attribútumaira pozícionált argumentumokkal, és kulcsszavas argumentumokkal is hivatkozhatunk. A mintaillesztés folyamata a következő sorrendben zajlik:

- Első lépésként meghívódik egy `isinstance(Alany, Class)`. Ezzel ellenőrizzük, hogy az *Alany* tényleg a mintában definiált osztály egy objektuma-e.
- Ha az első lépés sikeres, utána minden lehetséges almintát kell kiértékelni, balról jobbra haladva. Ha ezek közül bármelyik sikertelen, a class pattern is sikertelennek bizonyul.
- A pozícionált argumentumok az osztálynak a `__match_args__` attribútuma alapján lesznek kiértékelve.
 - Ha több pozícionált argumentumot adtunk meg, mint ami a `__match_args__` hossza, vagy ha az osztálynak nincs ilyen attribútuma, akkor **`TypeError`** keletkezik.
 - Az *i*-edik argumentum az mindig az *Alany*nak a `__match_args__[i]` attribútumára lesz illesztve.
- A kulcsszavas argumentumok attribútumként lesznek megkeresve az *Alany*on. Amennyiben nincs az *Alany*nak ilyen attribútuma, a mintaillesztés sikertelen. Különben az argumentum mintája az attribútum értékére lesz illesztve.

`__match_args__`

Ha egy osztály támogatni szeretné a pozícionált argumentumokkal való mintaillesztést, akkor definiálnia kell egy `__match_args__` nevű osztály-attribútumot. Az attribútum értékének pedig egy egyszerű, stringeket tartalmazó listának/tuple-nek kell lennie, ami sorrendben tartalmazza a támogatott pozícionált argumentumokat. Erősen ajánlott az osztály konstruktorának az attribútum listáját követni, hiszen maga a minta használata is egy konstruktorhívásra hasonlít (lásd 3. kódrészlet).

3.2.2. Literal Pattern

A literal pattern segítségével nem az *Alany* struktúrájára, hanem az értékére szabhatunk korlátokat. Segítségével emulálható a már más nyelvekben ismert *switch/case* szerkezet. Az *Alany* és a literal pattern értékének az összehasonlítása a Python általános egyenlőség-vizsgálatával történik. ($x == y$)

A lehetséges értékek, a minta nevéből is adódóan, a Python literálok lehetnek. Fontos itt is megjegyezni, hogy a minták nem tartalmazhatnak kifejezéseket, így tehát formátum stringek, rangok, stb. nem használhatóak.

További kivételt képez a **True**, **False**, **None** hármas. A félreértések elkerülése végett, ha a minta értéke **True**, akkor csak **True** értékű alanyra lesz sikeres a mintaillesztés (pedig pl. **True** == 1). Azonban, 1 értékű mintára viszont illeszkedik egy **True** értékű *Alany*.

3.2.3. Or Pattern

A nevéből is adódóan, az or pattern működése hasonló a Pythonban ismert **or** kulcsszó működéséhez. Segítségével egybefoghatunk több mintát, melyekből, ha egyre sikeres a mintaillesztés, akkor az egész or pattern sikeresnek bizonyul.

Változó hozzárendelés az or patternen belül csak úgy lehetséges, ha az összes minta ugyanahhoz a változóhalmazhoz rendel hozzá értéket. Vagyis, amennyiben van két mintánk, Q , P , és ezeket összefogjuk egy or patternbe: $Q \mid P$, akkor ha Q az két változóhoz, u -hoz és v -hez rendel hozzá értéket, akkor P -nek is pontosan ugyanazokhoz kell hozzárendelnie: u -hoz és v -hez.

Használatához a kívánt mintákat soroljuk fel, a 'l' karakterrel elválasztva. A minták sorrendben, balról jobbra fognak kiértékelődni.

3.2.4. Hozzárendelő minták

Míg az eddig tárgyalt minták fő feladata az *Alany* értékének és struktúrájának a vizsgálata volt, a hozzárendelő minták feladata ezzel szemben csak az *Alany* egy tetszőleges névhez való hozzárendelése. Az összes mintára igaz, hogy egy minta vagy egy megszorítást fejez ki, vagy egy értéket köt változóhoz, de nem mindkettőt.

Kivéve az **AS Pattern**-t, ami pont ezt a célt szolgálja. Segítségével egyszerre lehet egy általános mintát definiálni, és egy névhez rendelni az *Alanyt*. Használatához elsőként megadjuk a használni kívánt mintát, utána az **as** kulcsszó után megadjuk a nevet, amihez a minta *Alanya* lesz hozzárendelve.

Mivel a hozzárendelő minták nem szabnak megszorításokat az *Alany* struktúrájára, és/vagy értékére, ezért ezek nyelvtanilag **tagadhatatlanok**. Ilyen mintának tekinthető a **Capture Pattern** illetve a speciális változata, a **Wildcard Pattern**.

A **Capture Pattern** formája egy név. A minta bármilyen értéket elfogad, és ehhez a névhez rendeli az értéket, mely egy lokális változóként fog funkcionálni (kivéve, ha ez a név *nonlocal* vagy *global*). Egy mintán belül nem lehet többször felhasználni ugyanazt a nevet hozzárendelésre. Ez a függvénydefinícióknál látott függvényparaméterekhez hasonlít, ahol is szintén elvárás, hogy a paraméterek listájában ne legyen ismétlődés.

A **Wildcard Pattern** egy speciális **Capture Pattern**, mely bármilyen értéket elfogad, azonban nem hoz létre új lokális változót. Sokszor előjöhethet az igény, főképp szekvenciák mintaillesztése során, hogy csak a fontos értékekre koncentráljon a minta, a többit hagyja figyelmen kívül. Ennek a megoldására hozták létre a **Wildcard Pattern**-t, ami nélkül sok esetben felesleges neveket kéne létrehozni, az amúgy sem felhasznált értékek lekötésére. Használatához a `'_'` karaktert kell használni, mely más, szerkezeti mintaillesztés használó programozási nyelvekben is ugyan erre a célra fenntartott speciális karakter.

```
match obj: # obj az Alany
    case Cat(color="Orange", weight = "a lot"): # Első elágazás
        give_food(obj, Lasagne())
    case Cat(color="Black" | "Gray"): # Alminták használata
        turn_around()
    case Cat(color = clr) if cat_affinity > 0: # Guard használata
        print(f"Its a {clr} cat!") # Binding használata
    case _: # "Tagadhatatlan" elágazás
        raise ValueError("Object is not a cat!!")
```

Kódrészlet 3: Összefoglaló példa a Python szerkezeti mintaillesztés használatára

4. fejezet

AST

Az AST (Abstract Syntax Tree) a programozási nyelvek fordítási folyamata során, legtöbbször a szintaktikai elemzés fázisa közben elkészített, belső reprezentációra használt adatstruktúra. [13]

A környezetfüggetlen nyelvtanok derivációs fáihoz (Concrete Syntax Tree) ¹ hasonlóan az AST is a forráskód nyelvtani szerkezetének a reprezentálására szolgál, azonban az AST nem tartalmaz minden apró nyelvtani szabályt, sok esetben összevon, vagy akár elhagy csúcsokat a derivációs fából.

A forráskód AST-vé alakítása közben általában csak a kód szerkezeti, és tartalmi információi kerülnek megőrzésre, a "felesleges" információk eldobódnak. Azonban a fa csúcsaihoz gyakran tartoznak extra, kiegészítő információk, melyek a fordítási folyamat későbbi fázisai közben kerülnek hozzáadásra. Nincsen "univerzális AST", programozási nyelv függő az AST-k pontos implementációja, azonban többnyire minden AST megőrzi a forráskódban található utasításokat, ezeknek a sorrendjét, változókat, ezeknek a típusát, deklarációjuk helyét, nevét, illetve operációkat, és azoknak az operandusait.

Könnyen kezelhető és tömör, így nem véletlen hogy a rengeteg kódelemző, és kódt-ranszformáló keretrendszer az AST-t használja a forráskód reprezentálására, manipulálására. [10] [5]

Ebben a fejezetben röviden bemutatom a Python beépített AST modulja által nyújtott lehetőségeket, illetve a fontosabb struktúrákat, melyeket a transzformációs keretrendszer vizsgál.

¹https://en.wikipedia.org/wiki/Parse_tree

4.1. Python AST

A Python azon programozási nyelvek közé tartozik, melyek beépítve támogatják a hozzáférést a saját nyelvük AST-jéhez. Ezáltal a Python forráskód transzformációhoz, és analízáláshoz nincs szükség egy harmadik féltől származó eszközre. A beépített Python AST modul [22] használatával hatékonyan és könnyen konvertálható bármilyen Python forráskód AST-vé, melynek a manipulációjához is számos eszközt biztosít a modul.

Az AST előállításának a legkönnyebb módja az `ast.parse()` függvény használata, melynek paraméterül adva a forráskód stringjét eredményként egy objektumokat tartalmazó fát kapunk, mely objektumok mind az `ast.AST` osztályból származnak. A modul nem csak a forráskód AST-vé alakítására szolgáltat lehetőséget, hanem az AST bájtkóddá, vagy forráskóddá alakítására is ad módot. Fontos kiemelni, hogy a visszafordítás során az AST felépítéséből adódóan néhány részlet a kódban átalakul, vagy elveszhet. Többek között a kommentek elvesznek, a több sorba rendezett utasítások egy sorba kerülnek, illetve a string literálokat jelző karakterek átalakulhatnak.

Az AST manipulációjához a modul két alaposztályt definiál, az `ast.NodeVisitor` és az `ast.NodeTransformer` osztályokat. Ezeket örököltetve létrehozhatunk osztályokat, melyeket felhasználva iterálhatunk végig egy AST fán. Nekünk kell definiálni, hogy mely csúcsokat érintsék, és ezeket hogyan dolgozzák fel. A legfontosabb különbség a kettő osztály között, hogy míg a **NodeVisitor** nem képes az eredeti fa módosítására, a **NodeTransformer** igen. Például, a **NodeVisitor** segítségével kigyűjthetjük a forráskódból az összes változó előfordulást, míg a **NodeTransformer** segítségével át is nevezhetjük ezeket. A transzformációs keretrendszer értelemszerűen a **NodeVisitor**-t használja a forráskód analízálására, és a **NodeTransformer**-t a transzformálásra.

A transzformációs keretrendszer a feltételes elágazásokat vizsgálja a kódban, és ezeket próbálja meg szerkezeti mintaillesztéssé transzformálni. Ehhez természetesen a forráskód AST-jét használja, ezért szerintem érdemes végig futni ezeknek az elemeknek az AST-beli reprezentációján, hogy a későbbiekben egyszerűbb legyen a transzformációs logika követése, megértése.

4.2. Vizsgált elemek

4.2.1. If node

Az analízálás szempontjából a legfontosabb AST objektum az **If node**. Egy feltételes elágazást reprezentál. Attribútumai:

- **test**: Az elágazáshoz tartozó feltételt tartalmazza.
- **body**: Egy utasításokat tartalmazó lista, melynek elemei az elágazás testét reprezentálják. Ez az a kódrészlet, mely a feltétel beteljesülése esetén fog lefutni.
- **orelse**: A további elágazásokat tartalmazza, rekurzív módon. Amennyiben a következő elágazás feltétel nélküli (azaz egy **else**: blokk), akkor a **body**-hoz hasonlóan egy utasítássorozatot tartalmazó lista. Utolsó elágazás esetén az értéke **None**.

A könnyebb kezelhetőség érdekében ezeket a csúcsoakat egy *Brancheket* tartalmazó listává konvertálom. Minden *Branch* tartalmazza az If node-hoz tartozó feltételt (test), illetve az elágazás testét (body). Így az **if ... elif ... elif ... else ...** szerkezeteket sokkal könnyebben, nem rekurzív módon tudom kezelni.

4.2.2. BoolOp node

Mivel a feltételek struktúráján alapszik a transzformáció végrehajthatósága, ezért szintén nagyon fontos elemek a feltételekhez kapcsolódó AST elemek. Az egyik legfontosabb ilyen elem a **BoolOp node**, melyek a boolean operációkat reprezentálják a logikai kifejezésekben. Attribútumai:

- **op**: A boolean operációt határozza meg. Lehetséges értékei értelemszerűen `ast.And()`, és `ast.Or()`, melyek természetesen az **and**, és az **or** kulcsszavakat reprezentálják.
- **values**: Lista, mely tartalmazza az operációhoz tartozó értékeket. Fontos, hogy az egymás után alkalmazott operációk egybeolvadnak.

Előfordulhat olyan, hogy a BoolOp node values attribútumában további BoolOp node-ok szerepelnek. Például az `a and b and (c or d or e)` kifejezésben egyértelmű, hogy a szülő BoolOp node values listája tartalmazni fog egy másik BoolOp node-ot, melynek az op attribútuma nem `ast.And()`, hanem `ast.Or()` lesz.

Zárójelek elhagyása esetén az operátorok precedenciájától függ, hogy mi kerül az AST "tetejére". Pythonban nagyobb precedenciája van az `and` kulcsszónak mint az `or`-nak², vagyis: `a or b or c and d` \iff `a or b or (c and d)`. Ez a produkált AST-ben azt jelenti, hogy az erősebb precedenciájú BoolOp node az a fában "lejjebb" lesz, mint a kisebb, vagyis zárójelezés nélkül a példában a külső BoolOp node op attribútuma `ast.Or()` lesz.

Semmi sem akadályozza meg a forráskód íróját abban, hogy "felesleges" zárójeleket tegyen a logikai kifejezéseibe, például: `a and b and (c and d and e)`. Világos, hogy ebben az esetben a zárójelezés logikailag felesleges, azonban az AST-ben ez az előző példához hasonlóan két BoolOp node lenne. Ez a kifejezések kezelését megnehezítené, ezért előfeldolgozás során ellenőrzöm, és javítom az ilyen típusú kifejezéseket.

4.2.3. Compare node

A feltételekben gyakran előfordul, hogy egy (vagy több) értéket hasonlítunk össze egy (vagy több) változóval. Ezért az összehasonlítások reprezentálásáért felelős **Compare node** is figyelemre méltó. Attribútumai:

- **left:** Az első felhasznált érték az összehasonlításban
- **ops:** Lista, mely tartalmazza a felhasznált operandusokat. Ez határozza meg az összehasonlítás módját. Lehetséges értékek többek között az `ast.Eq()`, `ast.Lt()`, `ast.Is()`, stb., melyek rendre az `a == b`, `a < b`, és az `a is b` kifejezések operandusait reprezentálják.
- **comparators:** Lista, mely tartalmazza az első felhasznált érték után következő értékeket.

²https://www.mathcs.emory.edu/~valerie/courses/fall10/155/resources/op_precedence.html

Bár a Pythonban lehetőségünk van több értéket egyszerre is összehasonlítani (pl. `1 <= a < 10`), az ilyen fajta összehasonlításokat nem lehet szerkezeti mintaillesztéssé alakítani, így a transzformálás során nem fordítok figyelmet az ilyen alakú kifejezésekre.

4.2.4. Match node

A szerkezeti mintaillesztés blokkot reprezentálja (vagyis a **match** kulcsszót). Attribútumai:

- **subject**: A mintaillesztés *Alanyát* tartalmazza.
- **cases**: Lista, mely a **match_case node**-okat tartalmazza. Vagyis a mintákat, és az általuk definiált elágazásokat tartalmazza.

Az elágazásokat a mintaillesztési blokkon belül a **match_case node**-ok reprezentálják, melyeknek az attribútumai:

- **pattern**: A mintát határozza meg, melyre az *Alany* értéke lesz illesztve. Lehetséges értékei **Match[Pattern]** osztályú, mintákat reprezentáló objektumok (lásd 4.2.5).
- **guard**: Egy logikai kifejezést tartalmaz, mely a sikeres mintaillesztés után értéklődik ki. Értéke lehet **None**.
- **body**: Hasonlóan az **If node**-ok body attribútumához, ez egy lista, mely azt a kódszegmenst reprezentálja, mely sikeres mintaillesztés esetén fog végrehajtódni.

A *Branchek* és a **match_case node**-ok közötti hasonlóságra érdemes figyelni. A transzformáció alapja a *Branchek* feltételei és a **match_case node**-okhoz tartozó (pattern, guard) leképezés megvalósítása, mely a később tárgyalt *Pluginok* feladata lesz.

4.2.5. Match[Pattern] node-ok

A 3.2. fejezetben tárgyalt mintákat reprezentálják. A **Literal Pattern**-t a **MatchValue** illetve a **MatchSingleton** objektumok testesítik meg az AST-ben. Egyetlen attribútumuk a *value*, mely a literál értékét tárolja. A **MatchValue** attribútumának lehetséges értékei a már említett Python literálok, melyeket egyenlőség alapján hasonlít össze az

Alany értékével. A `MatchSingleton` attribútum lehetséges értékei pedig a már említett **True**, **False**, **None** hármas, melyet identitás alapján hasonlít össze az *Alany* értékével.

Az **Or Pattern**-t a **MatchOr** objektum reprezentálja. Egyetlen attribútuma a *patterns*, mely egy lista, ami tartalmazza az összevont mintákat. Amennyiben bármelyik mintára sikeres a mintaillesztés, az egész minta sikeresnek bizonyul.

A **Class Pattern**-t pedig a **MatchClass** objektum képviseli. Attribútumai:

- **cls**: Az illesztendő osztályt definiálja.
- **patterns**: Lista, mely sorrendben tartalmazza a vizsgálandó pozícionált attribútumokra illesztendő mintákat.
- **kwd_attrs**: Lista, mely a vizsgálandó kulcsszavas attribútumokat tartalmazza.
- **kwd_patterns**: Lista, mely a vizsgálandó kulcsszavas attribútumokra illesztendő mintákat tartalmazza, a **kwd_attrs** által definiált sorrendben.

A mintaillesztés folyamata az *Alany* osztályának, és a mintában definiált *cls* attribútumnak az összehasonlításával kezdődik. Utána a mintában megadott attribútumok kerülnek mintaillesztésre. A folyamat bővebben ki van fejtve a 3.2.1. fejezetben.

Mindhárom **hozzárendelő mintát** az AST-ben a **MatchAs** objektum képviseli. Két attribútuma a *pattern*, és *name*, melyek rendre egy mintát, és egy változónevet reprezentálnak. Ezeknek a lehetséges értékei definiálják, hogy melyik mintát reprezentálja a *node*:

- **AS Patternt** fejez ki, ha sem a *pattern*, sem a *name* attribútuma nem **None**.
- **Capture Patternt** fejez ki, amennyiben csak a *pattern* értéke **None**.
- **Wildcard Patternt** fejez ki, ha mind a *pattern*, mind a *name* attribútum értéke **None**.

Mivel a feltételes elágazásokban, ha történik is lokális változók létrehozása, azok nem feltétlenül az *Alanyhoz* kapcsolódnak, ezért a hozzárendelő mintákat csak az **else**: blokkok transzformációjánál használom.

5. fejezet

Transzformációs rendszer

A forráskód átalakítását a transzformációs keretrendszer vezérli. A rendszer tervezésekor nagy hangsúlyt fektettem a későbbi bővíthetőség lehetőségére. Arra is figyelmet fordítottam, hogy a rendszer könnyen beágyazható legyen más, kód-transzformációs keretrendszerekbe. A rendszer két fő részből áll: Az első, a forráskód analizálásáért felel, a második pedig magáért a transzformáció lebonyolításáért. Az analizálást biztosító rendszer (innen-től *Analyzer*) plugin felépítést ¹ használ: minden felismerendő minta tulajdonképpen egy plugin, melynek feladata a forráskód AST-jében fellelhető minták felismerése, és transzformációja. A bővíthetőség ezáltal adott, hiszen csak egy új plugint kell megvalósítani amennyiben egy új mintát szeretnénk definiálni. Az *Analyzer* feladata, hogy a definiált mintákkal felismertesse az inputjaként kapott AST-t, elvégzi a szükséges ellenőrzéseket, és elmenti, hogy a kapott input mely node-jai transzformálhatóak, és mely pluginokkal.

A transzformációt lebonyolító rendszer (innen-től *Transformer*) feladatai közé tartozik többek között a forrásfájlok beolvasása, AST-vé alakítása, az *Analyzer* inicializálása, és a transzformált kód összefésülése a forrásfájlokkal.

A rendszer jelenleg önállóan, külön programként funkcionál, azonban mivel a transzformációs logikát az *Analyzer* és a pluginok tartalmazzák, ezért ezek egyszerűen beágyazhatóak más rendszerekbe.

¹[https://en.wikipedia.org/wiki/Plug-in_\(computing\)](https://en.wikipedia.org/wiki/Plug-in_(computing))

5.1. Forráskód transzformáció

A transzformációs keretrendszer önálló használata nagyon egyszerű: A program kötelező argumentumként várja az átalakítandó forráskód elérési útvonalát. Ez lehet egy egyszerű *.py* fájl is, illetve akár egy teljes Python projektet tartalmazó mappa útvonala. Mindkét esetben alapértelmezetten a transzformált kód az eredeti fájl "mellé" lesz helyezve, *transformed* – [*source*] néven. Azonban van lehetőség megadni egy opcionális argumentumot, az *-i/* – *-inline* jelzőt. Használatával elkerüljük a fájlok új munkaterületre való másolását, és felülírjuk a forráskódot a transzformált kóddal.

A transzformálás logikájának a természetéből adódóan (lásd 6. fejezet) előfordulhat, hogy a transzformált kód emberi szemmel kevésbé olvasható, maga a transzformálás feleslegesnek tűnhet. A probléma szubjektív mivolta miatt nincsen rá univerzális megoldás, azonban személyre szabhatóak különböző "limitekkal", melyeket az *Analyzer*-re és a *Transformer*-re szabunk ki. Ezt a *config.ini* szöveges fájlban található konfigurációs lehetőségekkel tehetjük meg. Ilyen "limit"-nek tekinthető például a *MinimumBranches* opció, mellyel megszabhatjuk, hogy egy feltételes elágazás legalább hány ággal kell hogy rendelkezzen ahhoz, hogy a transzformáció megtörténjen. További limitek például a maximum ismételhető kódsorok száma, vagy a "csúnya lapítás" engedélyezése. A későbbi fejezetekben minden konfigurációs opció be lesz mutatva.

A Python AST felépítése [22] miatt minden *.py* fájl külön egységként kell kezelni. Ezért első lépésként a fájlstruktúrából rekurzívan ki kell gyűjteni a megadott útvonalon lévő összes *.py* fájl. Ezután fájlonként inicializálni kell egy új *Transformer*-t, melynek a feladata lesz a kapott fájl transzformálni. Ez a folyamat nagy projektek esetén időigényes lehet, azonban a fájlok nem függenek egymástól transzformációs szempontból, ezért ezt egyszerre több szálon ² is el lehet végezni.

A következőkben a transzformációs folyamat kerül bemutatásra. Ez a folyamat csak egy fájlra értendő, a fájlok kigyűjtése, másolása, és a többszálúság vezérlése ennek nem része. A transzformációs folyamatot mutatja be az 5.1. ábra, melyben kiemeltem a keretrendszer 3 fő részét.

A folyamat fázisai:

²[https://en.wikipedia.org/wiki/Multithreading_\(computer_architecture\)](https://en.wikipedia.org/wiki/Multithreading_(computer_architecture))

- A *Transformer* a transzformálandó forrásfájl elérési útját várja bemenetként. A fájlt beolvassa, és átalakítja AST-vé, melyből kiválogatja a transzformálandó kódrészleteket.
- Az *Analyzer* előfeldolgozás során egy könnyebben kezelhető formára hozza a kapott kódrészletek AST-jét (lásd 4.2.1 - 4.2.2). Ezeket átadja a pluginoknak, melyek megállapítják, hogy található-e a kódrészletben az általuk definiált minta. A kapott eredményekből lesz eldöntve, hogy lehet-e a kódrészletet transzformálni. (Erről bővebben a 6. fejezetben)
- Miután az *Analyzer* végzett, annak eredményét felhasználva a *Transformer* végzi el az átalakítást a pluginok segítségével.
- Amennyiben egy adott kódrészlet nem transzformálható, a *Transformer* rekurzívan elkezd további részekre bontani, és ezeket újra átadni az *Analyzer*-nek.
- A Python AST visszafordítása során nem garantált, hogy az eredeti forráskódot kapjuk vissza. [22] Ezért a *Transformer* csak azokat a kódrészleteket fordítja vissza, melyekről biztosan lehet tudni, hogy változtak. Ezeket a megváltozott kódrészleteket fésüli össze tehát a forráskóddal, melyeket aztán a kimeneti fájlba ír.

5.2. Transformer

A transzformációs folyamat első lépéseként a *Transformer*-t inicializáljuk egy forrásfájlal. Feladatai közé tartozik az *Analyzer* inicializálása, eredményeinek feldolgozása, a forrás- és eredmény fájlok kezelése, szintaktikai ellenőrzése.

SZINTAKTIKAI ELLENŐRZÉS

Előfordulhat, hogy maga a forrásfájl nem egy nyelvtanilag helyes *.py* fájl, mely esetben a transzformáció nem lehetséges. Ekkor a *Transformer* egy log fájlba értesíti a felhasználót, hogy a forrásfájlban hol, és milyen szintaktikai hibát talált. Önellenőrzésként továbbá, a már összefésült, transzformált fájlokat is ellenőrzi szintaktikai hibákért. Így garantált, hogy a rendszer nem produkál helytelen forráskódot. A log fájl generá-

lás a *config.ini* fájlban konfigurálható az *AllowTransformerLogs* és az *AllowOutput* opciók segítségével.

REKURZÍV TRANSZFORMÁLÁS

Amennyiben egy elágazás nem transzformálható, a *Transformer* megpróbálja az elágazás testét, külön egységként, rekurzívan transzformálni (lásd 6.2.1). Ez nem minden esetben produkál olvashatóbb kódot, ezért ez a lehetőség is a *config.ini* fájlban konfigurálható a *VisitBodiesRecursively* opcióval.

FORRÁSKÓD INTEGRITÁS MEGŐRZÉSE

Az AST-vé alakítás során bizonyos kódrészletek átalakulhatnak, vagy akár el is veszhetnek. Ezért a *Transformer* feladata, hogy minimalizálja azon kódrészleteket, melyeket AST-ről fordít vissza. Minden transzformált kódrészletről tudjuk, hogy az eredeti forráskódban hanyadik sorban kezdődnek. Ennek felhasználásával tudja a *Transformer* összefésülni az eredeti forráskódot a transzformálttal. Sorról sorra másolja át az eredeti fájlt a transzformált fájlba, és amennyiben egy olyan sornál tart, amiről tudja, hogy transzformált kódrészlet, a forráskód helyett a visszafordított AST kódrészletből másol. A transzformált kódrészletben előfordulhatnak kommentek, melyek szintén elvesznek AST-vé alakítás során. Ezeket sajnos nem lehetséges az "eredeti helyükre" visszamásolni, azonban van rá mód hogy megőrizzük őket. A *config.ini* fájlban bekapcsolható a *PreserveComments* opció, melynek hatására a *Transformer* a Python tokenize [24] modul segítségével kigyűjti a forráskódban található kommenteket, melyekről összefésülés során megállapítható, hogy egy módosult kódrészletben találhatóak-e. Ha igen, a *Transformer* a kódrészlet elé másolja ezeket a kommenteket.

DIFF FÁJLOK GENERÁLÁSA

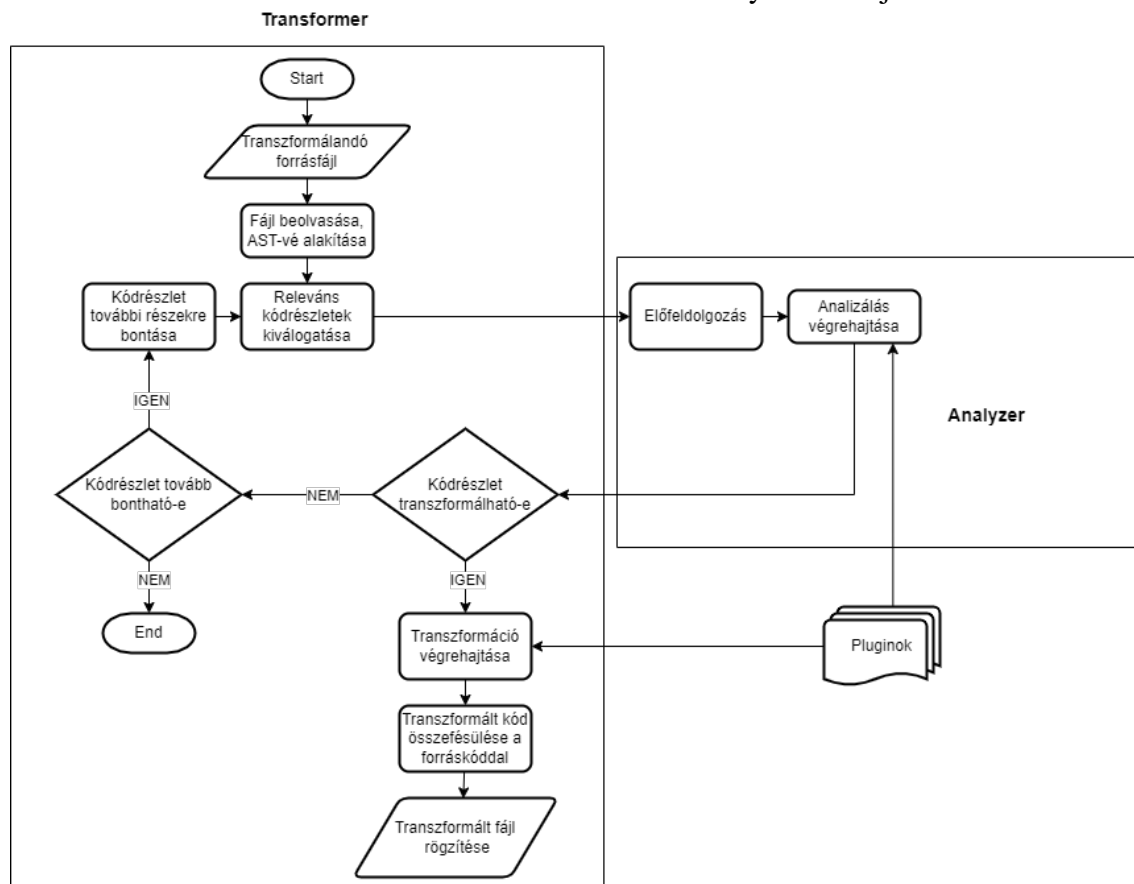
Kódtranszformáció esetén hasznos lehet a transzformált kód, és a forráskód összehasonlítása. Erre a célra megfelelő eszköz lehet egy diff ¹ fájl generálása. A nevéből is adódóan egy diff fájl tartalmazza a két összehasonlított fájlnak a különbségeit. Ezeket oly módon tárolja, hogy emberi szemmel és számítógéppel is könnyen feldolgozható legyen. Segítségével lehet patch ² fájlokat is generálni, melyek szöveges fájlok algoritmikus frissítését segítik elő. A *config.ini* fájlban a *GenerateDiffs* opció bekapcsolásával

¹<https://en.wikipedia.org/wiki/Diff>

²[https://en.wikipedia.org/wiki/Patch_\(computing\)](https://en.wikipedia.org/wiki/Patch_(computing))

a *Transformer* a fájl transzformációja után generál egy diff-fájlt a Python difflib [23] moduljának a segítségével, a forráskódot és a transzformált kódot felhasználva.

5.1. ábra. Forráskód transzformáció folyamatábrája



5.3. Analyzer

Az *Analyzer* feladata az inputként kapott forráskód AST-jének a vizsgálata. Előfeldolgozás során kigyűjti az AST-ből a vizsgálandó kódrészeket, melyeket további, könnyebben kezelhető egységekre, *Branch*-ekre bont. A *Transformer* az *Analyzer*-től kapja meg a transzformálható elágazások listáját, és a transzformációk módját. Az *Analyzer* feladatait itt csak összefoglalom, az 6. fejezetben mélyebben is ki lesz fejtve az AST vizsgálatának, átalakításának a folyamata.

PLUGINOK KEZELÉSE

Egy elágazás csak akkor transzformálható, ha minden *Branch*-ét felismeri legalább egy Plugin. Ezért az *Analyzer* feladata elágazásonként minden branchet átadni az

összes pluginnak. Amennyiben egy elágazás tartalmaz olyan *Branch*-et, melyet nem ismer fel egy plugin sem, akkor az elágazás biztosan nem transzformálható. Ellenkező esetben további ellenőrzések szükségesek.

ALANY MEGÁLLAPÍTÁSA

A szerkezeti mintaillesztés egyik alapja az *Alany*. Az *Alany* értéke lesz a mintákra illesztve, melyről a 3.2. fejezetben már volt szó. Az *Analyzer* feladata, hogy ellenőrizze, hogy minden *Branch*-et felismerő plugin ugyanazt tekinti-e a minta *Alany*-ának. Amennyiben minden *Branch* felismerhető legalább egy pluginnal, azonban nincsen közös *Alanyuk*, az elágazás nem transzformálható.

"LAPÍTÁS"

Bizonyos esetekben az egymásba ágyazott elágazások is transzformálhatók szerkezeti mintaillesztésre. (Tehát nem csak a *Transformer* által, rekurzívan) Azonban a rendszer felépítése miatt ezeket a pluginok csak akkor képesek felismerni, ha az egymásba ágyazást "kilapítjuk", ami az *Analyzer* feladata. Ennek hatására azonban kódsorok ismétlésére kényszerülhet a rendszer, ami sok esetben kerülendő. Ezért a lapítás konfigurálására a *config.ini* fájlban van lehetőség az *AllowFlattening*, *CodeRepetitionAllowed*, *MaxRepeatedLines*, és az *AllowUglyFlattening* opciók segítségével. Bővebben az egymásba ágyazott elágazásokról és a "lapításról" a 6.2. fejezetben lesz szó.

5.4. Pluginok

A keretrendszer talán legfontosabb, és mégis legkisebb részei, a pluginok feladata a *Branch*-ek feltételei és a *Case*-ek mintái közti leképezések megvalósítása. Feladatuk egy minta felismerése, lehetséges *Alanyainak* a meghatározása, illetve a transzformációja.

Minden mintához tartozik egy *Alany* halmaz, mely tartalmazza, hogy mely *Alanyokat* felhasználva lehetséges a minta transzformálása. Ezeket felhasználva határozza meg az *Analyzer* a teljes elágazás *Alanyát*.

Vannak minták, melyek tartalmazhatnak almintákat, ezért a pluginok hozzáférhetnek egymáshoz. Így egy egyfajta szülő-gyerek kapcsolat jöhet létre a minták között, ahol a szülő minta tartalmazhat egy vagy több almintát, melyeket a szülő felhasználhat az *Alany*

meghatározására, és a transzformációk elvégzésére. Továbbá lehet definiálni "komplex" mintákat is, melyek hozzáférhetnek a szülőmintáikhoz, azokat akár módosíthatják is. Ez olyan esetekben szükséges, amikor az alminták összevonhatóak egy, közös mintává.

6. fejezet

Megvalósított átalakítások

A transzformáció megértéséhez célszerű kiemelni a feltételes elágazások és a szerkezeti mintaillesztés közötti különbségeket. Tudjuk, hogy a feltételes elágazások *Branchek*-ből állnak, melyekhez rendre tartozik egy logikai kifejezés (feltétel), illetve egy kódszegmens, mely a **feltétel teljesülése esetén** fog lefutni. A szerkezeti mintaillesztés pedig egy *Alany*-ból, és *Case*-ekből áll, mely *Casek* pedig egy mintát és egy guardot (plusz feltételt) definiálnak, továbbá hozzájuk is tartozik egy kódszegmens, mely abban az esetben fut le, **ha az Alany a mintára illeszkedik, és a guard értéke igaz**.

Ebből következik, hogy **minden feltételes elágazás transzformálható** egy vele ekvivalens szerkezeti mintaillesztéssé. Hiszen minden *Branch* **feltétele megadható egy Case guardjának** (lásd 4. kódrészlet). Ebben az esetben bármit megadhatunk *Alany*-nak, és az összes mintát definiálhatjuk egy hozzárendelő mintának. Mivel a hozzárendelő minták tagadhatatlanok, ezért csak a guardban szerepelő logikai kifejezés értéke fogja eldönteni, hogy melyik *Case* kódszegmense fog lefutni.

```
if number == 0:
    print("Its zero!")
elif 1 == number:
    print("Its one!")
else:
    print("Default answer!")
```

⇒

```
match number:
    case _ if number == 0:
        print("Its zero!")
    case _ if 1 == number:
        print("Its one!")
    case _:
        print("Default answer!")
```

Kódrészlet 4: Szemmel láthatóan felesleges, azonban nyelvtanilag és logikailag helyes transzformáció

Világos, hogy ez a fajta transzformáció "csúnya": nem javít az olvashatóságon, sőt, felesleges indentációt, és nyelvtani "szemetet" ad hozzá a kódhoz. Természetesen a keretrendszer nem így végzi el az elágazások transzformációját, viszont ehhez szükségszerűen definiálnia kell egy leképezést, amely képes az elágazások feltételeiből egy vele ekvivalens mintát képezni.

Tegyük fel, hogy ez megvan, a *Branch*-ekből képesek vagyunk *Case*-eket gyártani, a *Branch*-hez tartozó kódszegmenst pedig minden gond nélkül átmásolhatjuk a *Case*-be. Azonban még mindig hiányzik egy dolog: az *Alany*. Nélküle nem tudunk mintaillesztést definiálni, azonban se a feltételes elágazásokhoz, se a mintákhoz nincsen semmilyen módon megadva, hogy mire szabnak feltételeket.

Mivel a feltételes elágazásokhoz ténylegesen nem tartozik extra információ, tulajdonképpen leírható egy *Branch*-et tartalmazó listaként, ezért joggal következik az, hogy a leképezésnek nem csak a mintát kell a feltételből megadnia, hanem valahogy a minta *Alany*-át is meg kell tudnia határoznia.

Röviden, szükség van egy f függvényre, $f(\text{bool_exp}) = (\text{pattern}, \text{subject})$ ahol $\text{eval}(\text{bool_exp}) = \text{True} \iff \text{match_pattern}(f(\text{bool_exp})) = \text{True}$, vagyis akkor és csak akkor illeszkedjen sikeresen az alany a leképezett mintára, ha az inputként kapott feltétel is igazra értékelődne ki az adott alany értékkel. A pluginok ezt a leképezést képviselik, mintánkként, így az alminták, és az összetett minták kezelése is egyszerűbb.

Azonban ezzel még nem adott a teljes transzformáció. Semmi sem garantálja azt, hogy az összes *Branch* leképezett *Case*-e ugyanazt a változót "tekintse" az *Alany*-ának, ami viszont elvárás a szerkezeti mintaillesztésnél, az összetett, és az almintákat tartalmazó mintákról nem is beszélve.

Az összetett, és az almintákat tartalmazó minták esetében maguk a pluginok felelnek az *Alany* ellenőrzéséért, míg az *Analyzer* azért felel, hogy a teljes szerkezeti mintaillesztés összes transzformált *Case*-e ugyanazt "tekintse" *Alany*-nak.

A következőkben bemutatom az implementált pluginokat, melyek, nem véletlen módon a 3.2. fejezetben bemutatott szerkezeti mintáknak felelnek meg. A félreértések elkerülése végett, ha a szerkezeti mintáról beszélek akkor a **Literal pattern** írásmódot fogom követni, míg ha pluginról, akkor **LiteralPattern** forma lesz használva.

6.1. Felismerhető minták

A felismerhető szerkezeti mintákat a pluginok reprezentálják. Feladatuk a logikai kifejezéseket az önmaguk által reprezentált szerkezeti mintává alakítani, illetve információt szolgáltatni arról, hogy a kapott kifejezésben mit "fogadnak el" *Alany*-ként. A logikai kifejezések vizsgálata az alábbi módon történik:

- A kifejezést megkapva a plugin elsőként megvizsgálja a kifejezés struktúráját, mely alapján eldönti, hogy lehetséges-e a transzformáció.
- Ha tartalmaz almintákat, azokat rekurzív módon vizsgálja, és a reprezentált szerkezeti minta logikája alapján le ellenőrzi az almintákat, és *Alanyaikat*.
- Amennyiben a transzformálás lehetséges, attribútumként elmenti a lehetséges *Alanyok* halmazát.

6.1.1. LiteralPattern

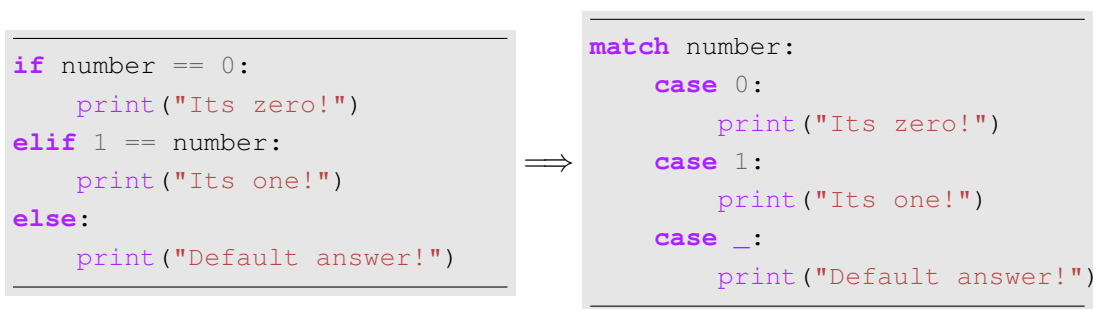
A legegyszerűbb, azonban talán a leggyakrabban felismert minta. Nem tartalmaz almintákat, és nem komplex minta. Segítségével felismerhetőek az olyan feltételek, melyek az *Alany* értékét hasonlítják össze egy konstans értékkel (lásd 5. kódrészlet).

Az ilyen feltételek felismerése, és az *Alany* meghatározása is meglehetősen egyszerű: `subject == constant` alakú feltételeknél egyértelműen megállapítható, hogy az összehasonlítás melyik oldalán található az *Alany*, és melyik oldalán a literál (lásd 4.2.3).

Továbbá, a mintaillesztés folyamata során pontosan ugyan ezen a módon kerül összehasonlításra az *Alany* és a minta által definiált literál értéke, ezért biztosak lehetünk a feltétel és a minta ekvivalenciájában (lásd 3.2.2.).

6.1.2. SingletonPattern

Az *Alany* értékét identitás alapján összehasonlító feltételeket ismeri fel, és transzformálja (lásd 6. kódrészlet). Az ilyen típusú feltételek felismerése is meglehetősen egyszerű: `subject is singleton` alakú feltételeknél egyértelmű az *Alany*, és a singleton értéke.



```

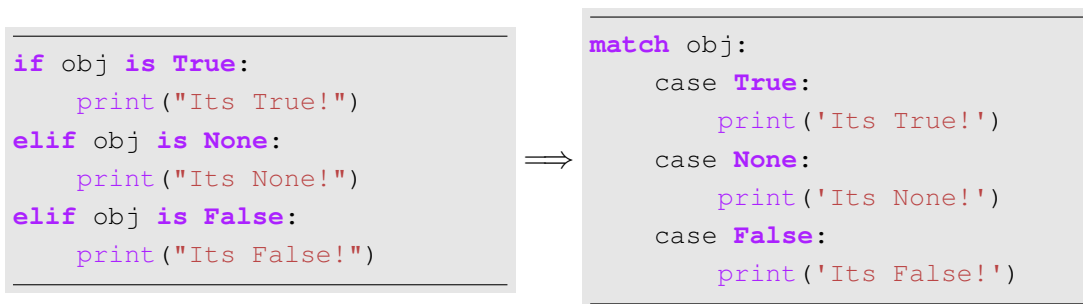
if number == 0:
    print("Its zero!")
elif 1 == number:
    print("Its one!")
else:
    print("Default answer!")
    
```

 \Rightarrow

```

match number:
    case 0:
        print("Its zero!")
    case 1:
        print("Its one!")
    case _:
        print("Default answer!")
    
```

Kódrészlet 5: A 4. kódrészlet transzformálása a LiteralPattern segítségével.



```

if obj is True:
    print("Its True!")
elif obj is None:
    print("Its None!")
elif obj is False:
    print("Its False!")
    
```

 \Rightarrow

```

match obj:
    case True:
        print('Its True!')
    case None:
        print('Its None!')
    case False:
        print('Its False!')
    
```

Kódrészlet 6: Példa a SingletonPattern által felismert feltételekre.

6.1.3. OrPattern

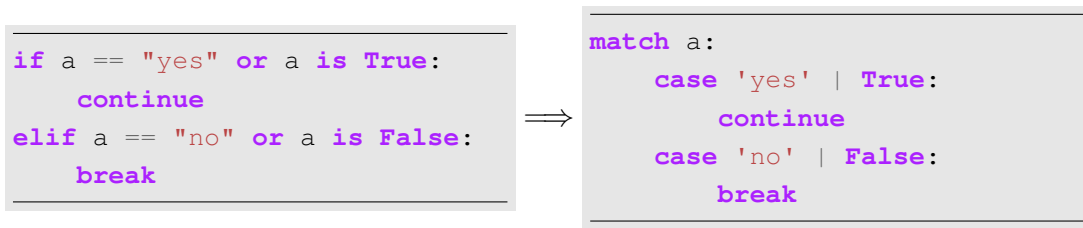
Talán a legegyszerűbb almintákat tartalmazó minta az OrPattern. Segítségével felismerhetők a $T1 \text{ or } T2 \text{ [or } T_n]^* \text{ }^1$ alakú feltételek. Az ilyen fajta feltételek akkor alakíthatók mintává, ha az összes T_n -re igaz, hogy azok önmagukban is felismerhető, és transzformálható minták, továbbá létezik *Alany* melyet közösen elfogadnak (lásd 7. kódrészlet).

A minta feladata átadni a többi plugin számára a lehetséges almintáit, majd az eredmények felhasználásával ellenőrizni, hogy létezik-e olyan *Alany*, melyet az összes alminták elfogad.

Mivel az Or Pattern azonos módon értékeli ki az almintáit, mint a vizsgált, **or** kulcsszóval adott feltételek (lásd. 3.2.3.), ezért biztosak lehetünk a feltétel és a minta ekvivalenciájában.

A 8. kódrészletben látható, hogy, ugyan az **or** kulcsszavakkal elválasztott kifejezések minták által felismerhetők, azonban nem ugyanazt tekintik *Alany*-nak, ezért a feltétel nem transzformálható.

¹Csillaggal a formális nyelvtanokhoz hasonlóan jelölöm a tetszőleges számú előfordulást



Kódrészlet 7: Példa az OrPattern használatára.

```
if a == "yes" or b is True:
    continue
elif a == "no" or b is False:
    break
```

Kódrészlet 8: Ebben az esetben az alminták nem ugyanazt tekintik *Alany*nak, ezért a feltétel nem transzformálható.

6.1.4. GuardPattern

A GuardPattern segítségével ismerhetőek fel a `T1 and T2 [and Tn] *` alakú feltételek. A boolean logikának megfelelően a logikai feltétel abban az esetben lesz igaz, ha az `and` kulcsszóval elválasztott `Tn` értékei mind-mind igazra értékelődnek ki. Ez kétféleképpen transzformálható mintává:

- Amennyiben az `and` kulcsszóval elválasztott `Tn`-ek együttesen egy szerkezeti megszorítást szabnak az *Alanyra*, akkor a GuardPattern (legalább egy része) összevonható egy, közös komplex mintává (lásd 6.1.5).
- Különben a GuardPattern csak akkor transzformálható, ha van legalább egy `Tn` mely egy felismerhető minta. Ebben az esetben a "maradék", nem felismert `Tn` a felismert mintához mint guard fog társulni.

Fontos, hogy amennyiben a Guard Pattern nem szülőminta, abban az esetben nem transzformálható. Példaként tekintsünk az alábbi feltételre: `a or b and c or d` mely a precedencia miatt ekvivalens az alábbival: `a or (b and c) or d`. Tegyük fel, hogy mind `a`, `b`, `d` felismerhető minták. Ebben az esetben `c` guard-ba kerülne. Első ránézésre tehát transzformálhatónak tűnik: `case a | b | d if c`. Azonban ez nyilvánvalóan nem ekvivalens átalakítás, hiszen az eredeti feltételben `c` értéke csak ak-

kor számított, amennyiben a és d értéke hamis, azonban a fenti mintában a guard értéke minden esetben ki lesz értékelve, és hatással lesz a mintaillesztés sikerességére.

Előfordulhat olyan eset, amikor a GuardPattern önmagában nem tudja eldönteni, hogy melyik Tn-t válassza a transzformált mintának, hiszen több lehetséges választás is van (lásd 9. kódrészlet). Ebben az esetben az *Analyzer* fogja eldönteni a választott mintát, méghozzá az *Alany* alapján. Vagyis a GuardPattern képes a választott *Alany* alapján eldönteni, hogy mi legyen a választott almintá, ami alapján a "maradék" a guard-ba kerül (lásd 10. kódrészlet).

Mivel a guard tulajdonképpen nem változtat a feltételeken, a választott minta pedig ekvivalens a transzformált feltétellel, ezért a GuardPattern is ekvivalens a teljes feltétellel.

```
if (a == 2) and (b == 4):
    something()
elif a == 6 and b == 7 and c():
    something_else()

⇒

match a:
    case 2 if b == 4:
        something()
    case 6 if b == 7 and c():
        something_else()
```

Kódrészlet 9: Példa a GuardPattern használatára, ahol a és b is lehetséges *Alanyok*. Az *Analyzer* ilyen esetben véletlenszerűen választ.

```
if (a == 2) and (b == 4):
    something()
elif (b == 8 or b == 9) and c():
    something_else()

⇒

match b:
    case 4 if a == 2:
        something()
    case 8 | 9 if c():
        something_else()
```

Kódrészlet 10: Példa a GuardPattern használatára, ahol a második elágazás rögzíti az *Alanyt*.

6.1.5. ClassPattern

Az első, és jelenleg az egyetlen almintákat tartalmazó, illetve komplex minta. Önmagában csak az `isinstance(obj, cls)` függvényhívást ismeri fel mint feltétel, viszont komplex mivolta miatt, amennyiben egy GuardPattern-en belül található meg, képes "átvenni" a GuardPattern további felismert almintáit.

A minta komplex mintaként definiálása szükségszerű, hiszen a Class Pattern (lásd 3.2.1.) az egyetlen olyan (dolgozatomban vett) minta, amely képes az *Alanya* egyszerre több szerkezeti feltételt szabni: Egyszerre képes az *Alanya* osztályát ellenőrizni, és az *Alanya* attribútumaira mintákat illeszteni. Ilyen fajta objektum feltérképezéshez mindenképpen a GuardPattern-nél látott alakot kell használni feltételes elágazások használata esetén.

Működése az alábbi módon történik: Ha a GuardPattern almintái között található egy felismert ClassPattern (vagyis egy `isinstance(obj, cls)` függvényhívás), akkor a GuardPattern "átadja az irányítást" a ClassPattern-nek: Megkapja az összes felismert almintát, melyeket képes "magába olvasztani". Ez egy egyszerű ellenőrzésen alapszik: Ha az al minta *Alanya* az a ClassPattern *Alanyának* egy attribútuma, akkor felvehető a mintába mint kulcsszavas attribútumhoz rendelt al minta. Például: `isinstance(S, C) and C.a == 2`. Világos, hogy a `C.a == 2` LiteralPattern által felismert al minta *Alanya* az C-nek egy attribútuma, ezért ez "beolvasztható" a ClassPattern-be.

A 11. kódrészleten jól látszik, hogy a ClassPattern képes bármilyen felismert mintát alkalmazni az attribútumokra, akár önmagát is. Ezzel, a példán is látható módon rendkívül növelhető a kód olvashatósága. Az `isinstance()` függvény második attribútumának megadható egy tuple, mely tartalmazhat több osztályt is, ilyenkor a feltétel akkor értéklődik ki igazra, ha az első attribútum bármelyik megadott osztálynak egy objektuma. Ezt a ClassPattern képes az Or Pattern segítségével kezelni (lásd 12. kódrészlet).

Észrevehető, hogy a ClassPattern nem támogatja a szerkezeti minta által definiált pozícionált argumentumokat. Ez természetes, hiszen nem lehetünk biztosak abban, hogy a vizsgált osztály implementálta-e az ehhez szükséges `__match_args__` osztály-attribútumot, továbbá a hagyományos feltételes elágazásokban ezeknek a használata nem is lehetséges.

Mivel a szerkezeti minta az `isinstance(obj, cls)` függvényhívást használja az objektum osztályának az ellenőrzésére, és a ClassPattern pedig pontosan ezt a függvényhívást ismeri fel, ezért a feltétel és minta ekvivalenciájában biztosak lehetünk. Továbbá az argumentumokra illesztett al minták is ekvivalensek a rájuk vonatkozó kifejezésekkel.

```
if isinstance(obj, A) and (obj.attr == 1 or obj.attr == 2):
    something()
elif ( isinstance(obj, B) and
      isinstance(obj.attr, A) and obj.attr.x == 3):
    something_else()
elif ( isinstance(obj, C) and
      isinstance(obj.attr, B) and isinstance(obj.attr.x, A)):
    something()
```

⇓

```
match obj:
    case A(attr = 1 | 2):
        something()
    case B(attr = A(x = 3)):
        something_else()
    case C(attr = B(x = A())):
        something()
```

Kódrészlet 11: Példa, mely bemutatja, hogy a ClassPattern bármilyen mintát képes alkalmazni az *Alany* attribútumaira.

```
if isinstance(obj, (A, B, C)):
    something()
elif isinstance(obj, (D, E)) and (obj.attr == 1 or obj.attr == 2):
    something_else()
```

⇓

```
match obj:
    case A() | B() | C():
        something()
    case D(attr = 1 | 2) | E(attr = 1 | 2):
        something_else()
```

Kódrészlet 12: ClassPattern működése, amennyiben több osztály is adott a feltételben.

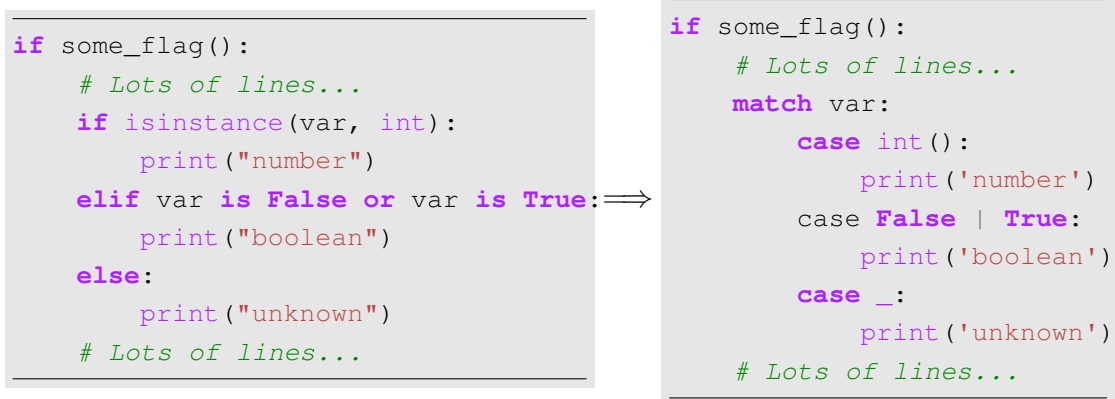
6.2. Egymásba ágyazott feltételes elágazások

A feltételes elágazások egymásba ágyazásával elérhető, hogy a beágyazott feltétel csak a szülő feltétel igazra értékelődése után legyen kiértékelve. Tehát egymás után kiértékelendő feltételeket fogalmazhatunk meg, továbbá lehetőséget biztosít arra, hogy a feltételek kiértékelése között is tudjunk tetszőleges kódot futtatni.

6.2.1. Rekurzív transzformálás

Az egymásba ágyazott feltételes elágazások legegyszerűbb transzformálási módja. Rekurzív módon kerülnek a beágyazott elágazások átalakításra, így bármilyen mélységben lévő kódszegmensek átalakításra kerülhetnek (lásd 13. kódrészlet).

Nem minden esetben produkál "szebb" kódot, viszont amennyiben egy nagy testű elágazás belsejében található, egy másik, transzformálható kódrészlet, azt csak így lehet átalakítani. Szubjektív "hasznossága" miatt ez a lehetőség a konfigurációs lehetőségek között kikapcsolható.



```
if some_flag():
    # Lots of lines...
    if isinstance(var, int):
        print("number")
    elif var is False or var is True:
        print("boolean")
    else:
        print("unknown")
    # Lots of lines...
```

 \Rightarrow

```
if some_flag():
    # Lots of lines...
    match var:
        case int():
            print('number')
        case False | True:
            print('boolean')
        case _:
            print('unknown')
    # Lots of lines...
```

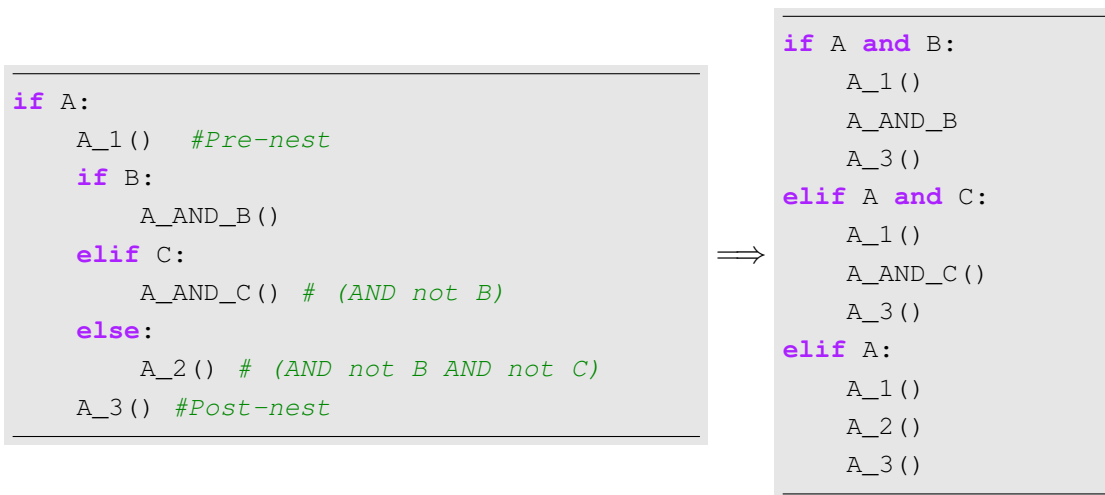
Kódrészlet 13: Példa az egymásba ágyazott feltételes elágazások rekurzív transzformálására.

6.2.2. "Lapítás"

Mivel a beágyazott elágazásnak a teste csak akkor fog lefutni, ha a szülő elágazás feltétele és a beágyazott elágazás feltétele is igazra értékelődik ki, ezért joggal mondhatjuk hogy a feltételek egymásba ágyazása ekvivalens az **and** kulcsszó használatával.

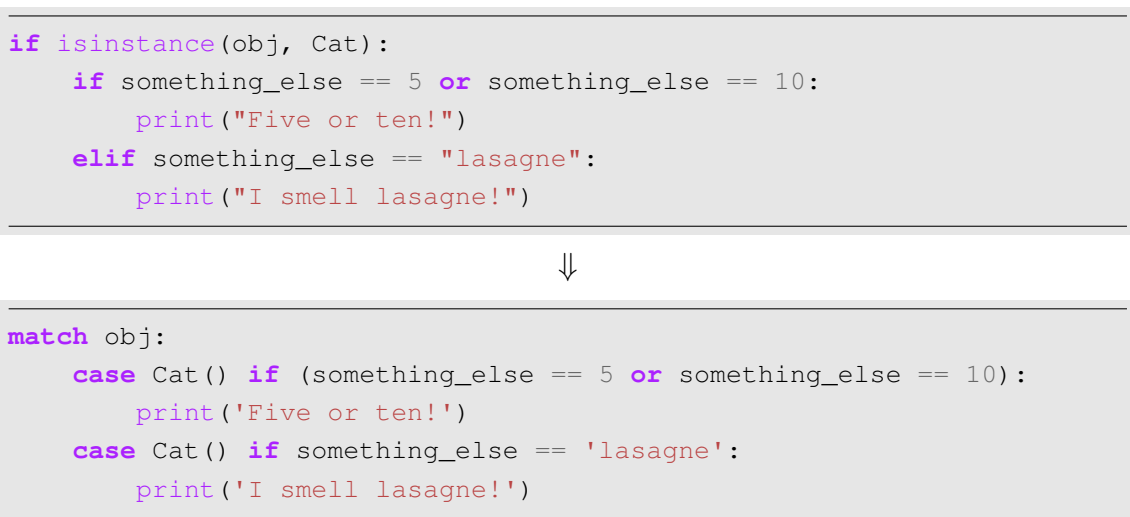
Ezt kihasználva lehetséges az egymásba ágyazott feltételes elágazásokat "kilapítani", azaz összevonni a szülő és a beágyazott elágazásokat. Minden beágyazott feltételhez hozzávesszük a szülő feltételét az **and** kulcsszóval (lásd 14. kódrészlet). Továbbá, ha a beágyazás előtt, vagy után volt kódszegmens a szülő elágazáson belül, azokat megismételve hozzá kell adni az összes, új elágazás elejére és végére, hiszen ezek minden esetben lefutottak, ha a szülő feltétel igaznak bizonyult. Vagyis a "lapítás" kódismétléssel járhat.

Nem minden esetben érdemes "kilapítani" az egymásba ágyazott elágazásokat, hiszen



Kódrészlet 14: Az egymásba ágyazott elágazások "lapításának" a logikája.

a kódismétlés általában elkerülendő. Továbbá világos, hogy az átalakított elágazások mind a GuardPattern által lesznek felismerve. Sok esetben az összevont feltételek nagy része ezáltal csak a minta guardjába kerülne, amely szintén kerülendő (lásd 15. kódrészlet). Azonban a komplex mintákat is csak a GuardPattern képes kezelni, ezáltal ideális esetben a "kilapított" elágazások egy komplex mintává alakíthatóak (lásd 16. és 17. kódrészlet). Mind a maximálisan ismételhető kódsorok száma, mind a "csúnya" lapítás engedélyezése a konfigurációs lehetőségekkel kontrollálható.



Kódrészlet 15: Példa egy felesleges "lapításra", melyet érdekesebb lenne rekurzív módon transzformálni.

```
if isinstance(obj, Cat):
    if obj.color == 'black' or obj.color == 'gray':
        turn_around()
    elif obj.color == 'orange' and obj.weight == 'a lot':
        give_lasagne()
    else:
        ignore_cat()
```



```
match obj:
    case Cat(color='black' | 'gray'):
        turn_around()
    case Cat(color='orange', weight='a lot'):
        give_lasagne()
    case Cat():
        ignore_cat()
```

Kódrészlet 16: Példa egy ideális "lapításra".

```
if isinstance(obj, Dog):
    give_treats(obj)
    if obj.breed == "golden retriever":
        command(obj, "Retrieve gold for me!")
    elif obj.breed == "german shepherd":
        command(obj, "Shepherd some germans for me!")
    command(obj, "Who's a good boy?")
```



```
match obj:
    case Dog(breed='golden retriever'):
        give_treats(obj)
        command(obj, 'Retrieve gold for me!')
        command(obj, "Who's a good boy?")
    case Dog(breed='german shepherd'):
        give_treats(obj)
        command(obj, 'Shepherd some germans for me!')
        command(obj, "Who's a good boy?")
```

Kódrészlet 17: Példa egy ideális "lapításra", mely kódismétlést tartalmaz.

7. fejezet

Tesztelés és kiértékelés

A fejlesztés során számos kódrészletet írtam tesztelés céljából. Amikor egy új funkció került a projektbe, vagy valamilyen változás történt, akkor ezeken a kódrészleteken futtattam a transzformációt. Ennek a folyamatnak a megkönnyítésére létrehoztam egy tesztelést végző szkriptet, mely a tesztmappában lévő *.py* fájlokra lefuttatja a transzformációt, és a lefordított fájlokat összehasonlítja, egy másik mappában lévő, referencia fájlokkal. A szkript jelez, ha a referencia és a lefordított fájlok nem egyeznek, illetve, ha nyelvtani hiba található az eredeti, vagy a lefordított fájlokban. Ez a fejlesztési folyamat során a tesztelést rendkívül megkönnyítette, hiszen nem kellett szemmel átnézni a tesztfájlokat.

A transzformációs rendszert továbbá leteszteltem több nyílt forráskódú projekten is: Instapy¹, discord.py², pylint³, keras⁴, django⁵, pandas⁶. A projektek pontos adatai megtalálhatóak a 7.1. táblázatban. A transzformációt minden projektre lefuttattam 1, 2, 4, 6, és 12 szálon, projektenként 3 alkalommal. Minden alkalommal lemértem a futásidőt⁷, illetve a maximális memóriahasználatot a futás során. Projektenként személyesen ellenőriztem legalább a transzformációk felét (a kisebb projekteknél az összeset), így összesen 360 db. transzformációt ellenőriztem az 597 db. végrehajtott transzformációkból, amely statisztikailag szignifikánsak tekinthető.

¹<https://github.com/InstaPy/InstaPy>

²<https://github.com/Rapptz/discord.py>

³<https://github.com/PyCQA/pylint>

⁴<https://github.com/keras-team/keras>

⁵<https://github.com/django/django>

⁶<https://github.com/pandas-dev/pandas>

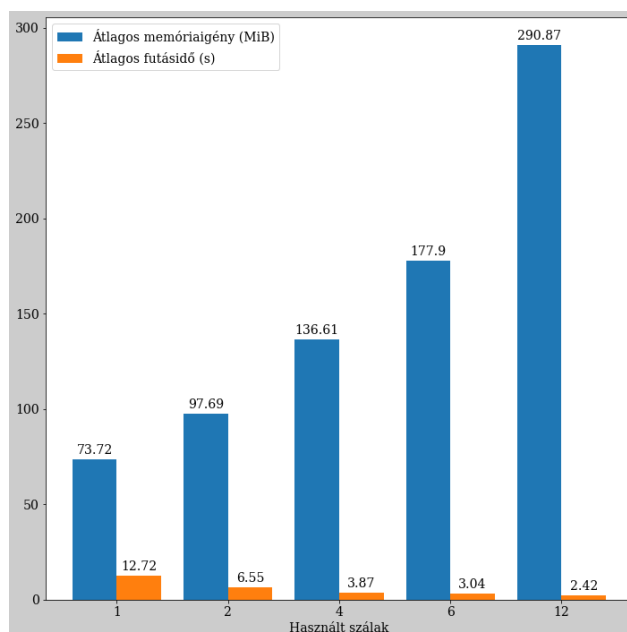
⁷A fájlok új munkaterületre való másolását nem számoltam bele a futásidőbe, hiszen ez a folyamat kizárólag a háttértár teljesítményétől függ.

Projekt	Vizsgált kódsorok	Vizsgált fájlok	Vizsgált node-ok	Transzformált node-ok
InstaPy	13 405 sor	38 db	1001 db	17 db.
discord.py	30 474 sor	144 db	2325 db	28 db.
pylint	58 896 sor	1276 db	3787 db	55 db.
keras	156 344 sor	687 db	7342 db	124 db.
django	326 138 sor	2129 db	9209 db	130 db.
pandas	358 075 sor	1355 db	12 496 db	243 db.

7.1. táblázat. A teszteléshez használt projektek adatai

7.1. Statisztikák

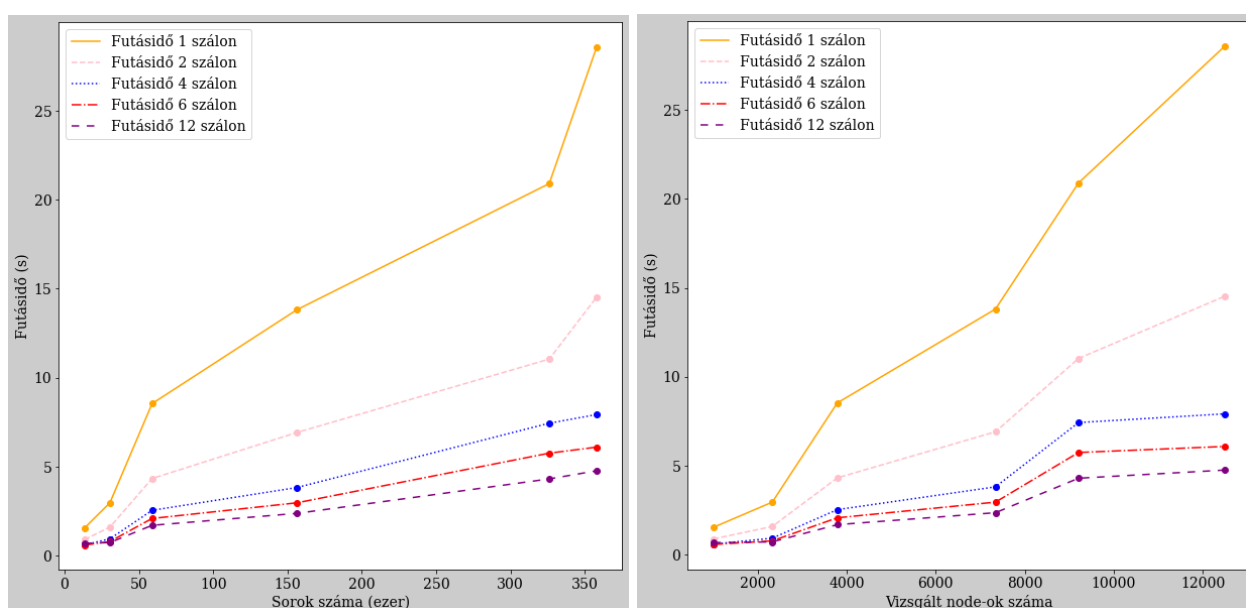
A tesztelést Windows 10 operációs rendszer alatt végeztem, egy AMD Ryzen™ 5 2600X, 6-magos processzort tartalmazó számítógépen. A transzformációs rendszer a többszálúság kezeléséhez a Python beépített ProcessPoolExecutor⁸ osztályát használja. Az említett processzor magonként 2 szálát képes kezelni, ezért az osztály alapértelmezetten maximum 12 szálát engedélyez. Ennél több szál használata csak a memóriaigényt növelné, a futásidőn nem változtatna, vagy akár növelhetné is. A fájlok több szálon való transzformálásával a futásidő nagymértékben javítható, ahogy az a 7.1. ábrán látszik. Több szál használata esetén azonban több fájlt is kell egyszerre a memóriában eltárolni, ezért a memóriaigény a használt szálakkal növekszik.



7.1. ábra. Az átlagos futásidő, és memóriaigény a használt szálak alapján csoportosítva.

⁸<https://docs.python.org/3/library/concurrent.futures.html#processpoolexecutor>

A futásidő a megfigyelések alapján legfőképp a vizsgált sorok számától függ. Ez nem meglepő, hiszen minden transzformálandó fájlt AST-vé kell alakítani, és transzformáció után pedig össze kell fésülni a transzformált AST eredményét a forrás fájlal, ami mindenképpen sorról-sorra történik. További faktor lehet még a vizsgált node-ok száma, azonban ez nem teljesen független a kódsorok számától. Ahogy az a 7.2. ábrán látszik, mindkét esetben, a várható varianciától eltekintve a futásidő egyenesen arányos a vizsgált kódsorok / node-ok számával. Megfigyelhető, hogy a használt szálak duplázása esetén nem feltétlenül feleződik a futásidő. Ez nagy valószínűséggel a használt háttértár teljesítménye miatt van.



7.2. ábra. Az átlagos futásidő és a vizsgált kódsorok / node-ok kapcsolata, a használt szálak alapján csoportosítva.

7.2. Talált hibák

A transzformált projektek ellenőrzése során felfedeztem néhány olyan transzformációt, melyek vagy nyelvtani hibát eredményeztek (mely esetben az egész fájl transzformációja meghiúsul a *Transformer* önellenőrzése során), vagy logikai hibát okoztak, vagy csak szimplán nem az elvártnak megfelelően működtek. A következőkben felsorolok pár érdekesebb ilyen transzformációt, és bemutatom a lehetséges javítás folyamatát.

HIBA A LAPÍTÁSBAN

A lapítás egész ötlete azon alapszik, hogy a szülő feltétel teljesülése esetén a beágyazott feltétel előtti, és utáni kódrészletek mindenképpen lefutnak. Azonban, ha a beágyazott kódrészlet ezt valahogyan megakadályozza, akkor a lapítás logikája nem fog működni. Erre fejlesztés során nem gondoltam, ezért ellenőrzés során találkoztam egy, a 18. kódrészletben bemutatott transzformációs hibával. Javítása viszonylag egyszerű: a lapítás ne legyen végrehajtva, amennyiben bármelyik beágyazott ág tartalmazza a **return**, **continue**, **break**, **yield** kulcsszavak bármelyikét, hiszen ezek a kulcsszavak megakadályozhatják, hogy a beágyazott elágazás utáni kódrészlet mindenképpen lefusson.

<pre>if isinstance(obj, Class) and a: if b: return obj.copy() return obj</pre>	⇒	<pre>match obj: case Class() if a and b: return obj.copy() return obj</pre>
--	---	---

Kódrészlet 18: Rövid példa mely bemutatja a lapítás hibás működését.

DINAMIKUS OSZTÁLY ELLENŐRZÉS

A szerkezeti mintákban sem kifejezéseket, sem változókat nem használhatunk, ezért szerkezeti mintaillesztés során a dinamikus osztály ellenőrzéseket csak a guard-ba tehetjük. Ezt azonban a ClassPattern nem vizsgálja, ezért ellenőrzés során az alábbihoz (19. kódrészlet) hasonló transzformációval találkoztam.

<pre>if isinstance(obj, type(var)): something() elif isinstance(obj, tuple(var2)): something_else() elif isinstance(obj, var3): something()</pre>	⇒	<pre>match obj: case type(var)(): something() case tuple(var2)(): something_else() case var3(): something()</pre>
---	---	---

Kódrészlet 19: Rövid példa mely bemutatja a ClassPattern hibás működését.

Világos, hogy az első két elágazás transzformációja nyelvtani hibát okoz, azonban viszonylag könnyen kiszűrhetőek: amennyiben a vizsgált `isinstance()` hívás második paramétere (tartalmaz) egy függvényhívás(t), az elágazás nem transzformálható.

Azonban a harmadik elágazás önmagában nem okoz nyelvtani hibát, és sajnos kiszűrni sem lehet, ugyanis AST-ben tulajdonképpen nincs különbség az alábbi két kifejezés között: `isinstance(obj, some_var)`, `isinstance(obj, SomeClass)`. A probléma tehát abból fakad, hogy csak futásidőben lehet megállapítani egy változóról, hogy mit tartalmaz: egy típust, ami felhasználható a Class Pattern-ben, vagy bármi mást. Az egyetlen lehetséges "megoldás" a problémára az lehet, hogy a változónak a nevéből a transzformációs rendszer megpróbálhat "tippelni". Legtöbb esetben Pythonban az osztályokat a *CamelCase* konvenció alapján szokták elnevezni, de ez nem garantált, így a hibára nincsen garantált megoldás.

EGYMÁSBA ÁGYAZOTT LISTÁK

Az `isinstance()` függvényhívás esetén lehetőségünk van egy tuple-ben több osztályt is definiálni az ellenőrzéshez. Azonban ebbe a tuple-be bármennyi, másik tuple-t is beágyazhatunk, amivel én fejlesztéskor nem voltam tisztában, így az alábbi (20. kódrészlet) transzformáció lehetségesnek bizonyult:

<pre>if isinstance(obj, (A, (B, C))): something()</pre>	⇒	<pre>match obj: case A() (B, C)(): something()</pre>
---	---	--

Kódrészlet 20: Rövid példa mely bemutatja, hogy a ClassPattern nem kezeli az egymásba ágyazott listákat.

A hiba javítása rendkívül egyszerű, hiszen a logikai kifejezések előfeldolgozásához hasonlóan (lásd 4.2.2. fejezet), itt is csak el kell hagyni a felesleges zárójeleket. Ez nem meglepő, hiszen a tuple használatával tulajdonképpen az `or` kulcsszó használatát "spóroljuk" meg, vagyis kiértékelés szempontjából az alábbi két kifejezés megegyezik: `isinstance(obj, (A, (B, C))), isinstance(obj, (A, B, C))`.

8. fejezet

Összegzés

A dolgozatomban elsőként általánosan bemutatam a problémát, melyre a refaktorálást adtam meg, mint lehetséges megoldás. Bemutattam, hogy milyen típusú problémákra lehet megoldás a forráskód transzformáció, illetve ismertettem néhány ilyen refaktorálást elvégző rendszert.

Ezek után részletesen bemutatam a probléma alapját, a szerkezeti mintaillesztést. Részletesen tárgyaltam az eredetét, hasznosságát, illetve működésének a logikáját és szabályait. Összefoglaló jelleggel tárgyaltam a Pythonban bevezetett szerkezeti mintaillesztés folyamatát, és az általa definiált, fontosabb mintákat.

Ismertettem a rendszer által – a forráskód strukturális reprezentációjára – használt AST-nek az eredetét és hasznosságát. Bemutattam a Python beépített AST modulja által nyújtott lehetőségeket, és részleteztem a fontosabb AST elemeket, melyeket a rendszer vizsgál.

Ezek után bemutatam az általam fejlesztett rendszer felépítését és képességeit, továbbá részleteztem a használatát, és ismertettem a rendszer lehetséges konfigurációs lehetőségeit. Majd a következőkben részleteztem a megvalósított átalakításokat, ahol ismertettem a transzformációs logikát, amely miatt a transzformáció hasznossága rendkívül szubjektív.

A rendszert több, kisebb-nagyobb nyílt forráskódú projekten is leteszteltem, mely során mind az időigényt, mind a memóriaigényt elemeztem. Az elvégzett transzformációkat ellenőriztem. Bemutattam a tesztelés közben felfedezett hibákat, melyekre részleteztem a javítás lehetőségeit.

Összességképpen, a korábbi fejezetekben bemutattam egy olyan transzformációs rendszert, mely Python forráskód refaktorálást végez, melynek célja az olvashatóság, és ezáltal a karbantarthatóság növelése. Tagadhatatlan, hogy az "olvashatóság", mint tulajdonság meglehetősen szubjektív, ezért a rendszer számos konfigurációs lehetőséget tartalmaz az egyéni igényekhez való viszonyulás érdekében. A rendszer akár egy mesterséges intelligenciát használó alrendszerrel is kibővíthető lehet, melyet megerősítéses tanulással lehetne megtanítani a nemkívánatos / felesleges transzformációk felismerésére.

A rendszer moduláris megvalósítása által viszonylag könnyen beágyazható más, AST alapú transzformációs rendszerekbe, illetve megvalósítható egy fejlesztői környezet pluginjaként is, mely képes lenne valós időben "javasolni" a feltételes elágazások transzformációját. Továbbá, a rendszer könnyen bővíthető további felismerhető szerkezeti mintákkal. Ilyen például a Sequence Pattern, mely a szekvenciák szerkezetére, és elemeire képes (al)mintákat illeszteni, vagy a Mapping Pattern, amely a Python dictionary-k feldolgozására ad új lehetőségeket [15].

Irodalomjegyzék

- [1] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 273–298, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [2] F. Geller, R. Hirschfeld, and G. Bracha. Pattern Matching for an objectoriented and dynamically typed programming language. Number 36 in Technical Report. Universitätsverlag Potsdam, 2010.
- [3] Paul black. static analyzers in software engineering. *crosstalk, the journal of defense software engineering*, pages 16–17, 2009.
- [4] Gábor Antal, Dávid Havas, István Siket, Árpád Beszédes, Rudolf Ferenc, and József Mihalicza. Transforming c++11 code to c++03 to support legacy compilation environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186, 2016.
- [5] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, and Youki Kadobayashi. Characterizing obfuscated javascript using abstract syntax trees: Experimenting with malicious scripts. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, pages 344–351, 2012.
- [6] Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, 11:477–502, 10 2002.
- [7] Brandt Bucher and Guido van Rossum. Structural pattern matching: Specification. PEP 634, 2020. Last accessed on 2022.04.16.

- [8] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 06 2000.
- [9] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. <http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial>, 01 1997.
- [10] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 668–673, 2010.
- [11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [12] Paolo G. Giarrusso. Optimizing and incrementalizing higher-order collection queries by ast transformation. 2020.
- [13] Tibor Gyimóthy, Ferenc Havasi, and Ákos Kiss. *Fordítóprogramok*. 01 2011.
- [14] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 946–957, New York, NY, USA, 2018. Association for Computing Machinery.
- [15] Tobias Kohn and Guido van Rossum. Structural pattern matching: Motivation and rationale. PEP 635, 2020. Last accessed on 2022.04.16.
- [16] Tobias Kohn, Guido van Rossum, Gary Brandt Bucher II, and Ivan Levkivskyi. Dynamic pattern matching with python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 85–98, 2020.
- [17] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477, 2018.

- [18] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.
- [19] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, USA, 1998.
- [20] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.
- [21] 2to3 - Automated Python 2 to 3 code translation. <https://docs.python.org/3/library/2to3.html>. Last accessed on 2022.04.16.
- [22] Python AST module documentation. <https://docs.python.org/3.10/library/ast.html>). Last accessed on 2022.04.16.
- [23] Python difflib module documentation. <https://docs.python.org/3/library/difflib.html>. Last accessed on 2022.04.16.
- [24] Python tokenize module documentation. <https://docs.python.org/3/library/tokenize.html>. Last accessed on 2022.04.16.
- [25] Luca Salucci, Daniele Bonetta, and Walter Binder. Efficient embedding of dynamic languages in big-data analytics. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 19–24, 2016.
- [26] Bart van Merriënboer, Dan Moldovan, and Alexander B. Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *NeurIPS*, pages 6259–6268, 2018.
- [27] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, 2001. Last accessed on 2022.04.16.
- [28] Daniel G. Waddington and Bin Yao. High-fidelity c/c++ code transformation. *Electronic Notes in Theoretical Computer Science*, 141(4):35–56, 2005. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

- [29] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, nov 1997.
- [30] Jingling Zhao, Kunfeng Xia, Yilun Fu, and Baojiang Cui. An ast-based code plagiarism detection algorithm. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pages 178–182, 2015.