Adaptacja algorytmów dopasowania do badania podobieństw programów

Adaptation of matching algorithms to test similarity of programs

Robert Kurcwald

WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

UNIWERSYTET OPOLSKI

Praca wykonana pod kierunkiem:

dra Sebastiana Bali

2021

Streszczenie

Ocena podobieństwa programów komputerowych to trudny problem. W literaturze naukowej problem ten pojawia się w ramach tematyki "software code clonning". Zagadnienie "software code clonning" ma bezpośredni związek z refaktoryzacją kodu. Potrafimy wyobrazić sobie dwie odmienne metodologie w projektowaniu rozwiązań szacujących podobieństwo programów. Pierwsza, wykorzystująca narzędzia sztucznej inteligencji i techniki oparte na uczeniu nadzorowanym. Druga oparta na definiowaniu kryteriów i metryk podobieństwa. Trudność pierwszego podejścia polega na przygotowaniu odpowiednio przetworzonych wejść i dużej ilości programów w danym języku programowania. Drugie podejście można realizować tworząc algorytmy heurystyczne korzystające z syntaktycznej struktury programów. Takie algorytmy powinny oceniać stopień podobieństwa na podstawie pewnych kombinacji zdefiniowanych kryteriów. W tej pracy próbujemy realizować drugie podejście na podstawie samodzielnie opracowanych algorytmów heurystycznych.

Słowa kluczowe: Porównywanie programów, ANTLR4, Algorytmy porównywania, Java, C++, Drzewa analizatorów składniowych, algorytmy heurystyczne

Abstract

Evaluating the source codes similarity is a complex problem. In Computer Science literature, the problem appears within the topic called "software code cloning". The issue of "software code clonning" is directly related to code refactoring. We can imagine two different methodologies applied to program similarity problem. First approach is based on Artificial Intelligence methods and techniques connected with supervised machine learning. The other one is based on defining criteria of the source codes similarities. The first approach requires preparation of special properly created input data for the particular programming language. The second approach can be realized by creating heuristic algorithms which use syntactic data structures. Such algorithms should evaluate program similarity by certain combination of defined criteria. In this thesis we realize the second approach using self developed heuristic algorithms.

Key words: Program matching, ANTLR4, Matching algorithms, Java, C++, Parse Tree, Heuristic algorithms

Spis treści

Rozdział 1

Wprowadzenie

Używanie szablonów czy generatorów kodu z pseudokodu jest coraz bardziej rozpowszechnione i wystarczy sprawdzić, że hostingi internetowe proponują coraz więcej takich rozwiązań by ułatwić użytkownikom zarządzanie swoimi zasobami, czy przyśpieszyć proces tworzenia stron. Jednakże to zagadnienie nie dotyczy tylko takich technik, ale również jest stosowane w formie ułatwienia przez programistów, którzy znajdują gotowe rozwiązania i adaptują je do swoich projektów. Według istniejących badań z 1998 roku [1], ilość podobieństwa we wszystkich badanych kodach wynosiła od 5% do 10% z tendencją rosnącą, natomiast badania z 2016 roku [6] raportują o wartościach oscylujących od 19% do nawet 59%. Postanowiliśmy zbadać podobieństwo na przykładzie studenckich programów napisanych w językach C++ oraz Java.
W pracy użyto generatora parserów ANTLR4 do generowania drzew syntaktycznych. Skorzystano do tego narzędzia z gramatyk stworzonych przez społeczność gromadzącą się wokół projektu zainicjowanego przez Terrence'a Parr'a. [4]

Ponadto opracowano heurystyki porównywania wartości oraz definicje stopni podobieństwa między programami, oparte o swoje przemyślenia.

1.1 Główne założenia pracy

Głównym celem pracy jest stworzenie schematu porównywania programów jedno-plikowych, który wskaże stopień podobieństwa plików wejściowych. Ta metoda pozwoli na pokazanie, w jakim stopniu studenci podczas pisania programów wzorują się na podanych przez prowadzącego pseudokodach i swojej wiedzy dotyczącej danego języka, a w jakiej skali używają gotowych szablonów i rozwiązań danego problemu.

Rozdział 2

Użyte technologie i narzędzia

Rozdział ten przedstawia technologie, ich opis, charakterystyki oraz przykłady potrzebne do zrozumienia dalszych aspektów pracy.

2.1 Java 8

Java 8 to kolejna wersja popularnego i funkcjonalnego obiektowego języka programowania, który został wydany w 1996 roku. W tym wydaniu twórcy zaimplementowali wyrażenia Lambda i metody wirtualnych rozszerzeń (ang. Virtual Extension). Dodatkowo zmienili system zabezpieczeń, przez co wrażliwe metody nie są wywoływane ręcznie, tylko poprzez precyzyjny mechanizm, który takie metody wyszukuje oraz uruchamia. Pełną listę zmian można znaleźć w dokumentacji [7] wydanej przez firmę Oracle.

2.1.1 Wybór języka do implementacji

Głównym powodem wyboru Javy, jako języka wiodącego był fakt, iż analizator składniowy opisany w sekcji 2.2.1 (ANTLR4), który będzie głównie używany w tej pracy, jest napisany w tym języku [3] przez co unikamy błędów składniowych czy ewentualnych błędów kompatybilności. Dodatkowo jest to rozbudowane narzędzie wraz z dużą ilością przykładów czy poradników, przez to próg zapoznania się jest relatywnie łatwy.

2.2 Generator analizatorów składniowych

Generatorem analizatorów składniowych (ang. parser generator) nazywamy narzędzie, które porównuje sekwencje znaków ze zdefiniowaną gramatyką, tworząc zwykle drzewa analizy składniowej (ang. parse tree). Te struktury pokazują w jaki sposób czytany jest kod przez daną jednostkę obliczeniową, a także przedstawia zależności pomiędzy kolejnymi typami (zwanymi synami) zdefiniowanymi w gramatyce. Analizatory składniowe zwykle używają gramatyk

bezkontekstowych (lub ich podklas), by przyśpieszyć generowanie oraz zwiększyć efektywność procesów podczas pracy.

2.2.1 ANTLR4

Another Tool for Language Recognition w wersji 4 to najnowsza, a zarazem najbardziej rozbudowana wersja narzędzia stworzonego przez profesora Uniwersytetu San Francisco Terence'a Parr'a w roku 1989 działająca na licencji BSD [9]. To oprogramowanie jest bardzo potężnym wieloplatformowym generatorem analizatorów składniowych (a także analizatorem leksykalnym), który zawiera moduły do odczytu, przetworzenia, wykonania czy przetłumaczenia danego ciągu znakowego na wygodną i uniwersalną formę zdefiniowaną gramatyką. Obecnie nad tym środowiskiem pracuje nie tylko Terence Parr, lecz cały sztab programistów, którzy są specjalistami w różnych dziedzinach, przez co biblioteki gramatyk bezkontekstowych są relatywnie aktualne, a także ogólnodostępne na profilu github projektu [4], co było jednym z powodów wybrania tego narzędzia celem wykorzystania w tej pracy. O popularności tego narzędzia i jego rozwiązań świadczą przykłady zastosowań przez największe firmy z branży IT i nie tylko:

- Twitter - wyszukiwarka tego popularnego portalu społecznościowego, która dziennie generuje ok 2 miliardów zapytań [8],
- Apache Hadoop - języki skryptowe (oraz SQL) generowane dla technologi serwerowej Hive and Pig [8],
- Oracle - narzędzia migracyjne oraz SQL Developer IDE [8],
- NetBeans - środowiska IDE parsujące język C++ [8],
- Hibernate - język HQL służący do obiektowo-relacyjnych zapytań do baz danych [8],
- Lex Machina - wyekstrachowywanie fragmentów tekstu z aktów prawnych [8].

Głównym zadaniem tego narzędzia jest wygenerowanie z gramatyki (formalnej definicji języka) analizatora składniowego, a następnie stworzenie struktury (drzewa analizy składniowej [ang. parse tree]), która pokazuje zależności między gramatyką, a wejściem. Dodatkowym plusem narzędzia jest to, że po wykonaniu poprzedniego etapu, od razu tworzy struktury pozwalające poruszać się po drzewie (tree walker).

Ciekawostką może być to, że od dłuższego czasu, ANTLR jest dodawany do wszystkich dystrybucji systemów Linux oraz OS X (firma Apple).

2.2.2 Generowanie drzew parsowania

Generowanie drzew parsowania przebiega w kilku kolejnych etapach dla każdego języka i przedstawia się w następujący sposób (Windows): [3]

1. Dodanie narzędzia ANTLR4 jako zmienna środowiskowa, by narzędzie było widoczne w każdej lokalizacji.

2. Stworzenie lub pobranie gramatyki odpowiadającej używanemu językowi (oficjalna strona projektu zlokalizowana na github.com [4]).

3. Uruchomienie w konsoli poleceń narzędzia ANTLR4 z nazwą i lokalizacją gramatyki (np. antlr4 Java8*.g4) w celu wygenerowania klas Lexera i Parsera.

4. Skompilowanie wszystkich wygenerowanych plików z rozszerzeniem wybranego języka (np. javac Java8*.java) celem utworzenia obiektów możliwych do wywołania.

5. Dodanie wygenerowanych elementów do projektu programu (np. Java+Maven w środowisku InteliIJ).

6. Wywołanie odpowiednio wygenerowanej klasy, tworzącej drzewo parsowania z pliku w następujący sposób: [Listing: 2.1, 2.2]

   (a) Stworzenie ciągów znaków (CharStreams) z danej lokalizacji.

   (b) Stworzenie Lexera dla danego ciągu znakowego z odpowiadającej językowi skompilowanej gramatyki.

   (c) Stworzenie CommonTokenStream, jako listy tokenów danej gramatyki wygenerowanej dla wyżej wygenerowanego Lexera.

   (d) Stworzenie Parsera dla danej listy tokenów.

   (e) Wygenerowanie drzewa parsowania z Parsera wybierając korzeń (dla Javy compilationUnit(), dla C++ translationUnit()).

7. Otrzymujemy plik wynikowy z rozszerzeniem .txt.

```
1  public void Cpp(String path,boolean saveToFile) throws
       ↪ IOException
2  {
3      charStreams= CharStreams.fromFileName(path);
4      CPP14Lexer cpp14Lexer = new CPP14Lexer(charStreams);
5      CommonTokenStream commonTokenStream=
6          new CommonTokenStream(cpp14Lexer);
7      CPP14Parser cpp14Parser = new CPP14Parser(commonTokenStream)
           ↪ ;
8          #Rozpocznij drzewo od zdefiniowanego
9          #korzenia (translationUnit dla CPP)
```

```
10      ParseTree parseTree = cpp14Parser.translationUnit();
11
12          #Wygeneruj drzewo w formie graficznej
13      generateTree("CPP",cpp14Parser,parseTree,2,saveToFile);
14          #Wygeneruj drzewo w formie pliku z rozszerzeniem .txt
15      saveToTxtFile("CPP",parseTree.toStringTree(cpp14Parser));
16  }
```

Listing 2.1: Kod źródłowy metody CPP w klasie GenerateTrees.java

```
1   public void Java(String path,boolean saveToFile) throws
        ↪ IOException
2   {
3       charStreams= CharStreams.fromFileName(path);
4       Java8Lexer java8Lexer = new Java8Lexer(charStreams);
5       CommonTokenStream commonTokenStream=
6           new CommonTokenStream(java8Lexer);
7       Java8Parser java8Parser=new Java8Parser(commonTokenStream);
8           #Rozpocznij drzewo od zdefiniowanego
9           #korzenia (compilationUnit dla Java)
10      ParseTree javaParseTree= java8Parser.compilationUnit();
11
12          #Wygeneruj drzewo w formie graficznej
13      generateTree("Java",java8Parser,javaParseTree,2,saveToFile);
14          #Wygeneruj drzewo w formie pliku z rozszerzeniem .txt
15      saveToTxtFile("Java",javaParseTree.toStringTree(java8Parser)
            ↪ );
16
17  }
```

Listing 2.2: Kod źródłowy metody Java w klasie GenerateTrees.java

Rozdział 3

Grafy i dopasowania

Rozdział ten zawiera teoretyczne zagadnienia związane z teorią grafów i porównywaniem, które są głównymi aspektami technicznymi wykorzystywanymi w pracy.

3.1 Definicje dotyczące grafów

Wyróżniamy dwa podstawowe rodzaje grafów:

- grafem nieskierowanym nazywamy parę $G = (V, E)$, gdzie V to zbiór elementów zwanych wierzchołkami, natomiast $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. E jest rodziną dwuelementowych podzbiorów wierzchołków V.

- grafem skierowanym nazywamy parę $G = (V, E)$, gdzie V oznacza to co poprzednio, a $E \subseteq \{(u, v) : u, v \in V, u \neq v\}$. W tym przypadku E jest zbiorem uporządkowanych par wierzchołków. Elementy z E nazywamy krawędziami grafu

3.2 Definicja struktury drzewiastej

W tym rozdziale wprowadzimy formalną definicję drzewa. Będziemy się inspirować definicją z deskryptywnej kolekcji zbiorów [5].

Niech $\mathbb{N}$ będzie zbiorem liczb naturalnych, który będziemy traktować jako alfabet. Przez ϵ będziemy oznaczać ciąg pusty czyli o zerowej długości nad alfabetem $\mathbb{N}$. Zamiennie będziemy oznaczać przez i lub (i) jednoelementowy ciąg liczb z $\mathbb{N}$. Znakiem $\cdot$ będziemy oznaczać operacje złączenia dwóch ciągów. Dla $u = (u_1, u_2, ..., u_k)$ i $v = (v_1, v_2, ..., v_i)$, złączeniem ciągów n i v jest ciąg $(u_1, u_2, ..., u_n, v_1, v_2, ..., v_i)$, co oznaczamy jako $u \cdot v$.

Niech A i B będą zbiorami ciągów nad alfabetem $\mathbb{N}$. Przez $A \cdot B$ oznaczamy zbiór $\{u \cdot v : u \in A$ i $v \in B\}$. A^i będziemy definiować indukcyjnie. $A^0 = \{\epsilon\}$, $A^1 = A$ oraz $A^{i+1} = A^i \cdot A$. Przez $*$

oznaczamy domknięcie Kleene'go definiowane równaniem $A^* = \bigcup_{i=1}^{\infty} A^i$.
Nośnikiem drzewa skończonego będziemy nazywać skończony podzbiór $T \subseteq \mathbb{N}^*$ spełniający następujące warunki:

1. $\epsilon \in T$ (drzewo ma korzeń)
2. Jeśli $w \cdot i \in T \Rightarrow w \in T$, gdzie $w \in \mathbb{N}^*$ (domknięcie na prefiksy)
3. Jeśli $w \cdot i \in T$ oraz $i > 0 \Rightarrow w \cdot (i-1) \in T$

Drzewem etykietowanym będziemy nazywać nazywać parę $\langle T, type \rangle$, taką że T jest nośnikiem drzewa skończonego, a *type* jest funkcją z T w zbiór $TYPES$, gdzie $TYPES$ jest skończonym zbiorem pewnych etykiet. Zbiór $TYPES$ jest skojarzony z gramatyką dla danego języka, a etykiety będziemy utożsamiać z symbolami nieterminalnymi gramatyki.

Kontekst drzewa etykietowanego $\langle T, type \rangle$ są to takie $A \subseteq T$, które spełnia następujące warunki:

- $\epsilon \in A$
- $w \cdot i \in A \Rightarrow w \in A$, $w \in N^*$

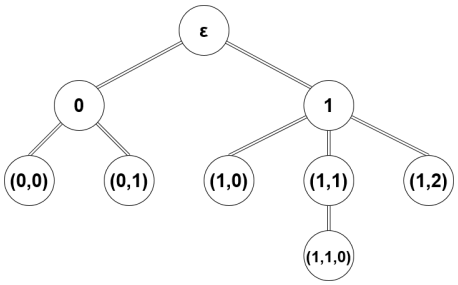
Wspólny kontekst drzew etykietowanych $\langle T, typet \rangle$**,**$\langle S, types \rangle$ to trójka $\langle A, B, f \rangle$, gdzie:

- A - kontekst drzewa etykietowanego $\langle T, typet \rangle$
- B - kontekst drzewa etykietowanego $\langle S, types \rangle$
- $f : A \to B$ jest bijekcją spełniającą:
  - Jeśli $f(u) = v$, to $|u| = |v|$, gdzie $|u|$ oznacza długość ciągu u
  - Jeśli $f(u \cdot i) = v \cdot j$, to $f(u) = v$
  - Jeśli $f(u) = v$, to $typet(u) = types(v)$

Poddrzewem etykietowanym - para $\langle X, typex \rangle$ jest poddrzewem drzewa $\langle T, type \rangle$ jeśli:

1. Jeśli $w \cdot i \in X \Rightarrow w \in X$, gdzie $w \in \mathbb{N}^*$
2. Jeśli $w \cdot i \in X$ oraz $i > 0 \Rightarrow w \cdot (i-1) \in X$
3. Jeśli $\forall_{(w \in X)}\ w \in T \wedge (w \cdot i \in T \Rightarrow w \cdot \in X)$
4. $\forall_{(\alpha,t)}\ \alpha = t \Rightarrow typex(\alpha) = type(t)$

Przykładem drzewa jest zbiór ciągów zobrazowany rysunkiem 3.1 oraz takim zbiorem:
$\{(\epsilon), (\epsilon, 0), (\epsilon, 1), (\epsilon, 0, 0), (\epsilon, 0, 1), (\epsilon, 1, 0), (\epsilon, 1, 1), (\epsilon, 1, 2), (\epsilon, 1, 1, 0)\}$



Rysunek 3.1: Przykład struktury drzewiastej w deskryptywnej kolekcji zbiorów

3.3 Heurystyczny algorytm wyszukiwania wspólnych kontekstów

Działanie algorytmu polega na porównywaniu pojedynczych węzłów obu drzew w tym samym momencie w celu znalezienia sum wspólnych kontekstów i wyznaczenia stopnia podobieństwa. Dla celów algorytmów wprowadzimy nowe operacje na elementach nośnika: *corp* oraz *tail*. Definiujemy je następująco:

- $corp(\epsilon) = \perp$ ($\perp$ - symbol błędu/sprzeczności)
- $corp(w \cdot i) = w$, dla $w \cdot i \in T$
- $tail(w \cdot i) = i$, dla $w \cdot i \in T$

Formalnie zakładamy, że $\perp \notin T$. W opisie algorytmu zakładamy, że $|(x)|$ to oznaczenie mocy zbioru x.
Poniżej przedstawiono pseudokod algorytmu wyszukiwania wspólnych kontekstów, [Pseudokod 1,2] który polega na porównywaniu wierzchołków w sposób niezależny od poziomu, lecz zależny od miejsca startu danego kontekstu (nazywanego korzeniem) do miejsca, gdzie typy się nie zgadzają (nie są równe).
Przebieg działania:
Początkowo na wejściu mamy dwa drzewa t oraz s. Bierzemy korzenie tych drzew i oznaczamy je jako ϵ. Blok zaczynający się w linii 2 odpowiada za sprawdzenie przynależności wierzchołków do zbioru odwiedzonych, a następnie porównanie wybranych par celem znalezienia podobieństwa typów. Do tego celu użyto zagnieżdżonych bloków w liniach 6,7,8,12,18,35.

- Blok z linii 6 (6-36): Warunek rozpoczynający tę sekwencję odpowiada zapytaniu "czy wierzchołek ma dzieci?".
- Bloki z linii 7 (7-34), 8 (8-17) oraz 12 (12-17): Bloki odpowiadające sprawdzeniu czy dziecko zostało odwiedzone, a także wybraniu takiego wierzchołka (dziecka), które nie było odwiedzone.

- Blok z linii 35 (35-36): Wykonanie skoku, gdy wierzchołek nie ma już więcej dzieci (jest liściem) do linii 13.
- Blok z linii 39-52, a także jego zagnieżdżenia odpowiadają za przechodzenie do innych wierzchołków w drzewie S dla danego wierzchołka w drzewie T, a następnie procedura jest powtarzana dla każdego kolejnego wierzchołka w drzewie T.
- Blok z linii 53-64 odpowiada przechodzeniu równoległemu, przez oba drzewa, do dzieci lub ojców i zaczęciu sprawdzania wszystkich możliwości od początku.
- Linia 64 odpowiada za zakończenie działania algorytmu i zwrócenie mocy zbiorów: Vis_1,T,Vis_2,S.

Dodatkowo w pracy przedstawiono zmodyfikowaną heurystykę wyszukiwania wspólnych kontekstów [Pseudokod 3,4], która różni się pierwszego wariantu, tym że rozpatrujemy wszystkich potomków w przeszukiwaniu drzewa S od lewej do prawej [Pseudokod 2: Linie 45-47] i omijaniu tych, które zostały odwiedzone w przeciwieństwie do pierwszego, który przeszukuje drzewo S od prawej do lewej celem znalezienia pierwszego nieodwiedzonego wierzchołka [Pseudokod 4: Linie 44-50]. Uproszczony schemat porównywania przedstawiono na Rysunku: 3.3.

Pseudokod 1: Wyszukiwanie wspólnych kontekstów p.1

```
   Input: T_1 = ⟨T, ftype⟩ T_2 = ⟨S, stype⟩
1  t := ϵ; s := ϵ; A_1 := ∅; A_2 := ∅; Vis_1 := ∅; Vis_2 := ∅;
2  if t ∉ Vis_1 and s ∉ Vis_2 then
3      if ftype(t) = stype(s) then
4          Vis_1 = Vis_1 ∪ {t}; Vis_2 = Vis_2 ∪ {s};
5          i := 0; j := 0
6          if t · i ∈ T then
7              if t · i ∈ Vis_1 then
8                  if t · (i + 1) ∈ T then
9                      i := i + 1
10                     t := t · i
11                     Go to line: 7
12                 else
13                     if corp(t) ∈ T then
14                         t := corp(t)
15                         Go to line: 6
16                     else
17                         Go to line: 64
18             else
19                 if s · j ∈ S then
20                     if s · j ∈ Vis_2 then
21                         if s · (j + 1) ∈ S then
22                             j := j + 1
23                             s := s · j
24                             Go to line: 19
25                         else
26                             if corp(s) ∈ S then
27                                 s := corp(s)
28                                 Go to line: 19
29                             else
30                                 Go to line: 64
31                     else
32                         t := t · (i)
33                         s := s · (j)
34                         Go to line: 3
35         else
36             Go to line: 13
```

Pseudokod 2: Wyszukiwanie wspólnych kontekstów p.2

```
36
37      else
38          if corp(s) · (tail(s) + 1) ∈ S then
39              s := corp(s) · (tail(s) + 1)
40              Go to line: 3
41          else
42              if corp(t) · (tail(t) + 1) ∈ T then
43                  t := corp(t) · (tail(t) + 1)
44                  s := copr(s) · (tail(s) + 1) index := 0
45                  while corp(s) · (tail(s) − index) ∉ Vis_2 do
46                      s := corp(s) · (tail(s) − index)
47                      index := 1
48                  Go to line: 3
49              else
50                  t := corp(t)
51                  s := corp(s)
52                  Go to line: 2
53  else
54      if corp(t) · (tail(t) + 1) ∈ T and corp(s) · (tail(s) + 1) ∈ S then
55          t := corp(t) · (tail(t) + 1)
56          s := corp(s) · (tail(s) + 1)
57          Go to line: 3
58      else
59          if corp(t) ∈ T and corp(s) ∈ S then
60              t := corp(t)
61              s := corp(s)
62              Go to line: 2
63          else
64              return : |(Vis_1)|, |(T)|, |(Vis_2)|, |(S)|
```

Pseudokod 3: Wyszukiwanie wspólnych kontekstów z indeksem głębokości p.1

```
   Input: 𝒯₁ = ⟨T, ftype⟩ 𝒯₂ = ⟨S, stype⟩
1  t := ϵ; s := ϵ; depth := 0; index := 0; A₁ := ∅; A₂ := ∅; Vis₁ := ∅; Vis₂ := ∅;
2  if t ∉ Vis₁ and s ∉ Vis₂ then
3      if ftype(t) = stype(s) then
4          Vis₁ = Vis₁ ∪ {t}; Vis₂ = Vis₂ ∪ {s};
5          i := 0; j := 0
6          if t · i ∈ T then
7              if t · i ∈ Vis₁ then
8                  if t · (i + 1) ∈ T then
9                      i := i + 1
10                     t := t · i
11                     Go to line: 7
12                 else
13                     if corp(t) ∈ T then
14                         t := corp(t)
15                         Go to line: 6
16                     else
17                         Go to line: 67
18             else
19                 if s · j ∈ S then
20                     if s · j ∈ Vis₂ then
21                         if s · (j + 1) ∈ S then
22                             j := j + 1
23                             s := s · j
24                             Go to line: 19
25                         else
26                             if corp(s) ∈ S then
27                                 s := corp(s)
28                                 Go to line: 19
29                             else
30                                 Go to line: 67
31                     else
32                         t := t · (i)
33                         s := s · (j)
34                         Go to line: 3
35         else
36             Go to line: 13
```

Pseudokod 4: Wyszukiwanie wspólnych kontekstów z indeksem głębokości p.2

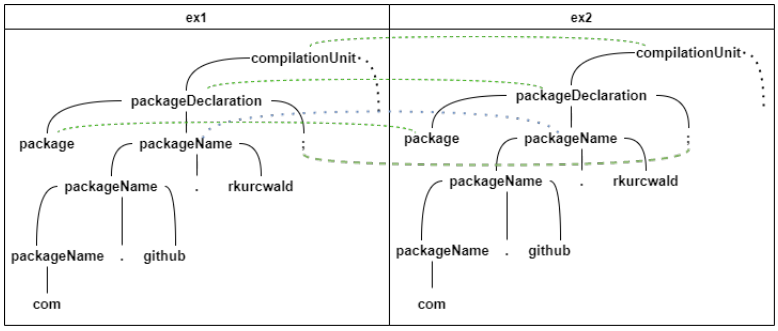
```
36
37      else
38          if corp(s) · (tail(s) + 1) ∈ S then
39              s := corp(s) · (tail(s) + 1)
40              Go to line: 3
41          else
42              if corp(t) · (tail(t) + 1) ∈ T then
43                  t := corp(t) · (tail(t) + 1)
44                  index := 0
45                  while corp(s) · index ∈ S do
46                      if corp(s) · index ∈ Vis_2 then
47                          index := index + 1
48                      else
49                          if corp(s) · index ∈ S then
50                              s := corp(s) · index
51                  Go to line: 3
52              else
53                  t := corp(t)
54                  s := corp(s)
55                  Go to line: 2
56  else
57      if corp(t) · (tail(t) + 1) ∈ T and corp(s) · (tail(s) + 1) ∈ S then
58          t := corp(t) · (tail(t) + 1)
59          s := corp(s) · (tail(s) + 1)
60          Go to line: 3
61      else
62          if corp(t) ∈ T and corp(s) ∈ S then
63              t := corp(t)
64              s := corp(s)
65              Go to line: 2
66          else
67              return : |(Vis_1)|, |(T)|, |(Vis_2)|, |(S)|
```

3.3.1 Schemat działania wybranego algorytmu porównywania

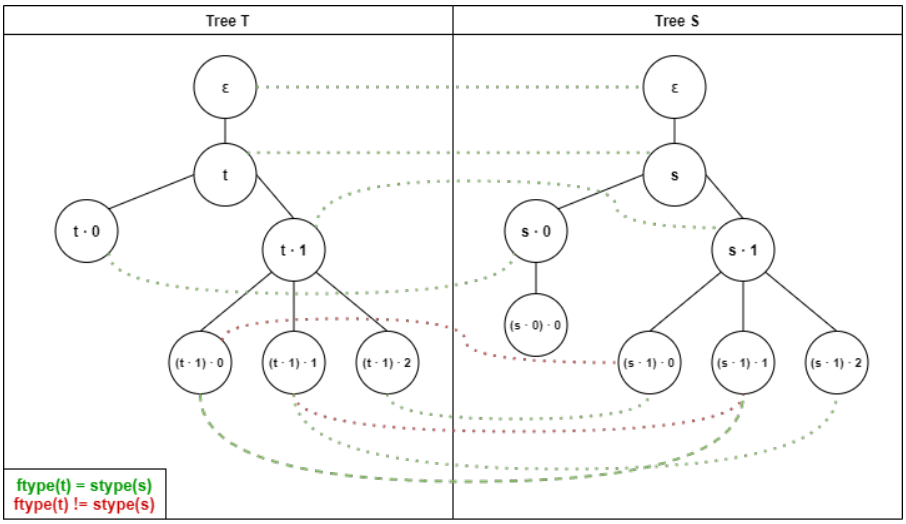
Algorytm wyszukiwania wspólnych kontekstów - w problemie porównywania to rekurencyjny schemat poruszania się po strukturze drzewa polegający na przejściu od korzenia do ostatniego liścia jednocześnie przez dwa drzewa celem znalezienia wspólnych poddrzew, zliczeniu ilości tych elementów i wyznaczenia uśrednionego stopnia podobieństwa ze wzoru: $P_{usrednione} = \frac{P_{AwB} + P_{BwA}}{2}$, gdzie P_{AwB} to stopień podobieństwa przedstawiający wielkość sumy wszystkich elementów podobnych poddrzew drzewa programu A w drzewie programu B, a P_{BwA} to stopień podobieństwa przedstawiający wielkość sumy wszystkich elementów podobnych poddrzew drzewa programu B w drzewie programu A. Oba podobieństwa obliczamy ze wzoru uogólnionego do postaci:

$$P_{XwY} = \frac{Liczba_{zaznaczonych\_jako\_odwiedzone\_wierzcholkow} - stala\_danego\_jezyka}{Liczba_{wszystkich\_wierzcholkow\_danego\_drzewa} - stala\_danego\_jezyka}.$$

W pseudokodzie [1,2] przedstawiającym schemat działania algorytmu, w zbiorach Vis_1 oraz Vis_2 są zawarte konteksty, które traktowane są jako odwiedzone.



Rysunek 3.2: Przykład porównywania drzew przez algorytm wyszukiwania wspólnych kontekstów



Rysunek 3.3: Przykład porównywania drzew przez algorytm wyszukiwania wspólnych kontekstów [Pseudokod: 1, 2]

3.3.2 Szacowanie złożoności obliczeniowej heurystycznego algorytmu porównywania

Oszacowanie złożoności dla wyżej wymienionych pseudokodów jest ciężkim zadaniem technicznym, przez użycie instrukcji *goto*. To samo może nastręczać problemu w dowodzeniu terminacji naszego algorytmu. Niestety nie udało się stworzyć wersji pseudokodu bez użycia tych instrukcji. Wygodnym w tym zadaniu byłoby posiadanie pseudokodu podzielonego na funkcje.
Pomocnym do sformułowania formalnego dowodu mogłyby być następujące spostrzeżenia bazujące na pseudokodzie 1 i 2:

- Algorytm porównuje (w linii 3) jedynie wierzchołki o tej samej odległości od korzenia.
- Jeśli para wierzchołków została dodana do (Vis_1, Vis_2) po pozytywnym sprawdzeniu warunku z linii 3 oraz aktualna para (t, s) nie leży w poddrzewach dodanych wierzchołków, to t i s nigdy więcej nie zostaną ustawione, jako wierzchołki poddrzewa aktualnych t i s.
- Po każdym wstawieniu pary (t, s) do zbiorów (Vis_1, Vis_2) porównujemy braci t z braćmi s. Takie porównania odbywają się co najwyżej raz, dla każdej pary, aż do momentu, gdy zostanie znaleziona para o tych samych typach lub pary zostaną wyczerpane.
- Po wyczerpaniu się par, o którym mowa w poprzednim punkcie, nigdy więcej nie porównujemy par które są potomkami ojców wspomnianych wierzchołków

Niech n będzie liczbą wierzchołków drzewa T i niech m będzie liczbą wierzchołków drzewa S. Z powyższych czterech warunków wynika, że instrukcje warunkowe naszego algorytmu mogą wykonać się $O((n+m)^2)$ razy bez pozytywnego rozstrzygnięcia warunku z linii 3. Natomiast pozytywnych rozstrzygnięć warunku z linii 3 może być co najwyżej $O(n+m)$. Razem daje to złożoność $O((n+m)^3)$.

Rozdział 4

Podobieństwo kodów źródłowych oraz ich charakterystyka

Porównywanie i wykrywanie podobieństwa programów jest trudne. Najłatwiejszy sposób wykrywania podobieństwa to porównanie programu w relacji 1:1 [2]. Niestety wyżej podany sposób nie sprawdza się dobrze przy różnego typu problemach opisanych w sekcji 4.1, w której przedstawiono potencjalne sposoby omijania tego typu programów i wykorzystywania cudzego kodu. Warto również zaznaczyć, że głównym celem nie jest wychwycenie małych stopni podobieństw, tylko fragmentarycznych oraz dużej skali, gdyż w środowisku akademickim zadania skonstruowane przez prowadzących są jednakowe dla całej grupy studentów, a głównie polegają one na rozszerzeniu lub przemodelowaniu podanych pseudokodów czy przykładowych programów.

- **Porównywanie programów** - jest to sprawdzanie podobieństwa programu drugiego do pierwszego w relacji skierowanej w celu oszacowania podobieństwa programów poprzez nadanie im pewnych ustalonych miar. Aby uzyskać relację symetryczną możemy badać odwrotną relację, czyli relację programu pierwszego do drugiego. Docelowo może służyć do tworzenia wszelkiego rodzaju programów antyplagiatowych.

- **Stopień podobieństwa programów** - jest to wyliczona wartość liczbowa lub procentowa (w naszym przypadku definiowana przez wzór z rozdziału 3.2) wynikająca z porównywania drzew wyprowadzenia (lub podrzew) programów. Zgodnie z algorytmem wyszukiwania wspólnych kontekstów, wynikiem końcowym jest liczba zmiennoprzecinkowa z precyzją do 4 miejsc znaczących, która reprezentuje stopień podobieństwa.

- **Drzewo wyprowadzenia** (ang. Parse Tree)(wymiennie: drzewa parsowania, PT) - jest to wynik analizy składniowej wejścia, który jest zgodny z podaną gramatyką. W tej pracy zostaną przedstawione drzewa parsowania wygenerowane przez narzędzie ANTLR, o którym więcej można przeczytać w rozdziale 2, dla języków Java w wersji 8 oraz C++ w wersji 14.

4.1 Typy problemów

W kolejnych podrozdziałach przedstawiono wieloletnie spostrzeżenia wynikające z przeglądania dużej ilości kodów przez promotora na przedmiocie "Algorytmy i struktury danych", które mają na celu zmniejszenie stopnia podobieństwa kodów programów.

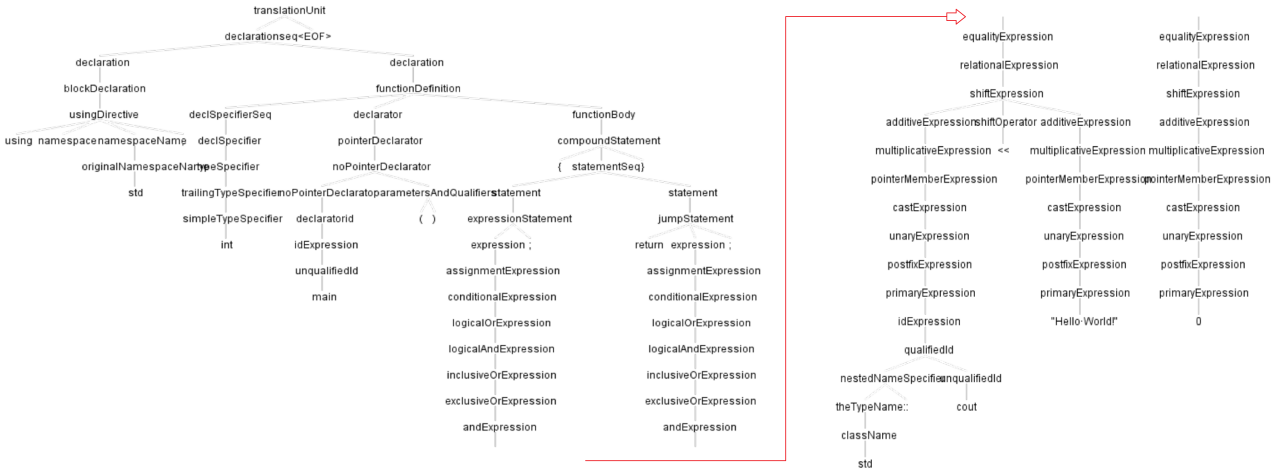
4.1.1 Dopisywanie kodu o dużej niezależności w porównaniu do oryginału

Podczas kopiowania programu osoba może dopisać redundantny kod, by utrudnić poprawne działanie programu sprawdzającego. Głównym pomysłem na rozwiązanie tego zagadnienia jest podzielenie kodu na poddrzewa wyprowadzenia i eliminacja tych poddrzew, które nie mają, żadnych odwołań do części właściwej kodu, co pozwoli nam na uzyskanie drzewa parsowania oryginalnego programu. Ponadto nie definiujemy wielkości, która jest uważana za dużą niezależność, gdyż dla każdego badania będzie to inna wartość, ale możemy podać przykłady, jakie rozumiemy pod tym pojęciem:

- Definicje zmiennych, metod czy klas, do których się nie odwołujemy
- Duplikowanie wywołań np. odwołanie kilkukrotne do przestrzeni nazw (Listing: 4.2)

```
1 #include <iostream.h>
2 using namespace std;
3 int main()
4 {
5     cout << "Hello World!";
6     return 0;
7 }
```

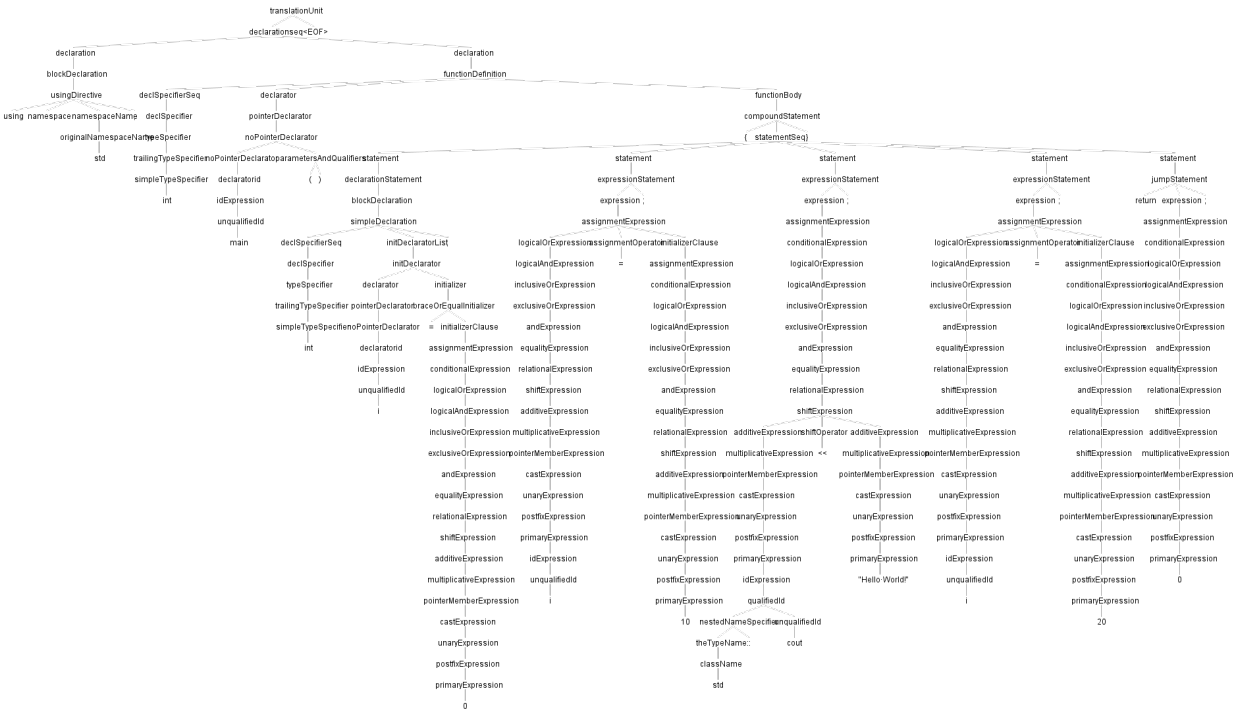
Listing 4.1: Kod źródłowy: HelloWorld.cpp



Rysunek 4.1: Drzewo parsowania wygenerowane dla HelloWorld.cpp (Listing: 4.1)

```
1  #include <iostream.h>
2  using namespace std;
3  int main()
4  {
5        int i=0;
6        i=10;
7        std::cout << "Hello World!";
8        i=20;
9        return 0;
10 }
```

Listing 4.2: Kod źródłowy: HelloWorld2.cpp (Redundantny kod)



Rysunek 4.2: Drzewo parsowania wygenerowane dla HelloWorld2.cpp (Listing: 4.2)

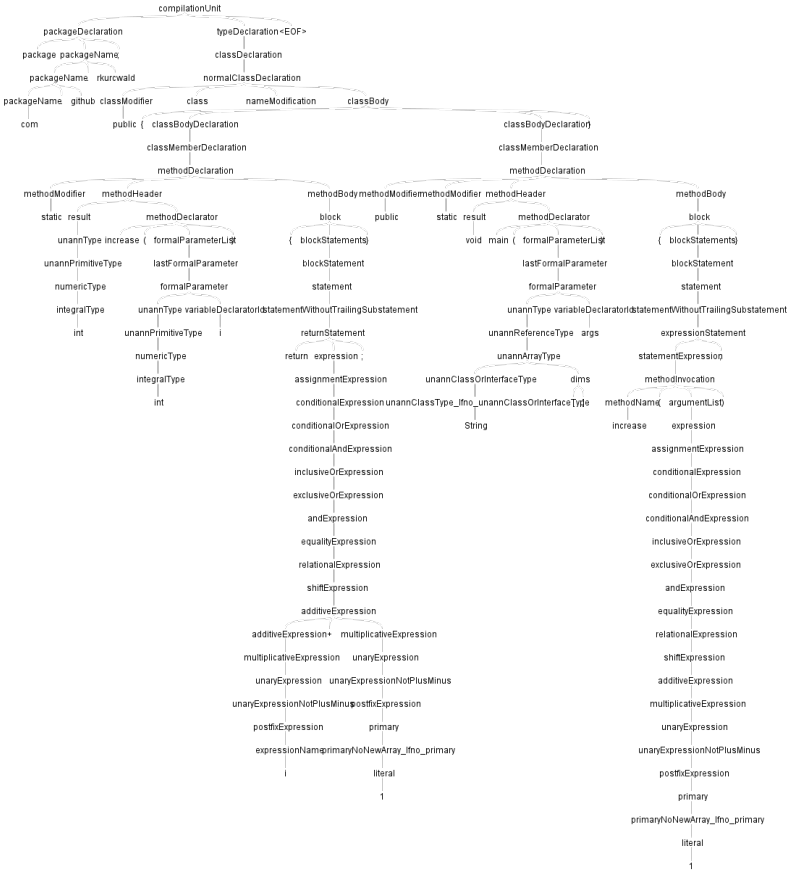
4.1.2 Modyfikacja nazw

Bardzo częstą praktyką podczas próby modyfikowania kodu jest zmiana nazw elementów kluczowych takich jak: zmienne, metody, klasy czy interfejsy. Rozwiązaniem tego typu problemu jest porównywanie struktury oraz typów, a nie samych nazw wyżej wymienionych elementów. Listingi [4.3] oraz [4.4] przedstawiają kody programów, które są identyczne pod względem struktury i ułożenia elementów, lecz inne pod względem nazewnictwa. Analizując rysunki [4.3] oraz [4.4] możemy wnioskować, że nie jest to duży problem dla algorytmu wyszukiwania wspólnych poddrzew, gdyż sama struktura drzew jest niezmienna, a jedyną różnicą są niektóre liście.

```
1  package com.github.rkurcwald;
2
3  public class nameModification
4  {
5      static int increase(int i)
6      {
7          return i+1;
8      }
9
10     public static void main(String[] args)
```

```
11          {
12                  increase(1);
13          }
14  }
```

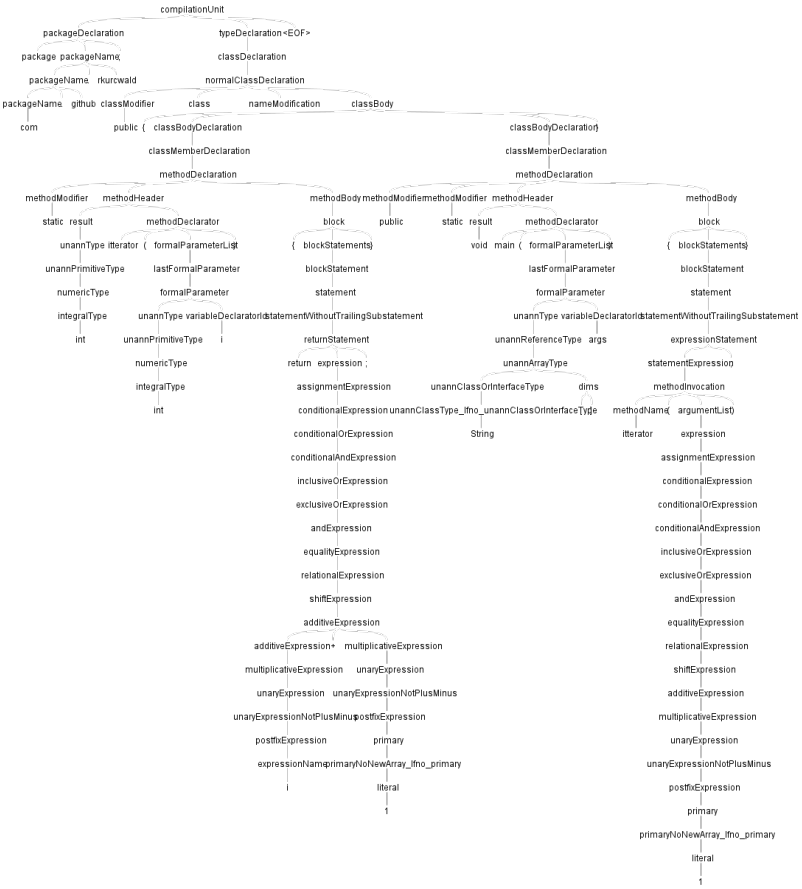
Listing 4.3: Kod źródłowy: nameModification.java



Rysunek 4.3: Drzewo parsowania wygenerowane dla nameModification.java (Listing: 4.3)

```
1  package com.github.rkurcwald;
2
3  public class nameModification
4  {
5      static int itterator(int i)
6      {
7          return i+1;
8      }
9
10     public static void main(String[] args)
11     {
12         itterator(1);
13     }
14 }
```

Listing 4.4: Kod źródłowy: nameModification2.java (Zmiana nazw)



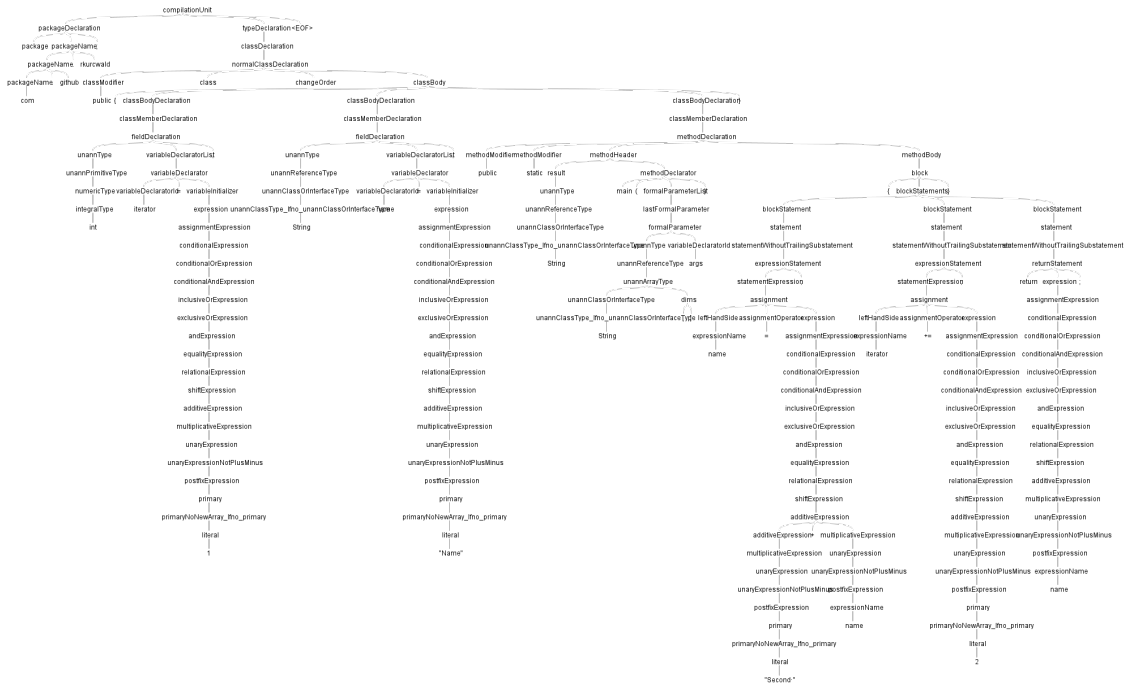
Rysunek 4.4: Drzewo parsowania wygenerowane dla nameModification2.java (Listing: 4.4)

4.1.3 Zmiana szyku programu

Zmiana kolejności elementów również jest bardzo częstą praktyką, która może zaburzyć działanie znalezienia stopnia podobieństwa. Rozwiązaniem tego problemu jest traktowanie synów (poddrzew), jakby te struktury nie musiały być w konkretnej kolejności, co niestety może w pesymistycznym przypadku prowadzić do znacznie zwiększonej złożoności obliczeniowej (trzeba przeszukać czy dany syn występuje w drugim kodzie na innej pozycji).
Listingi [4.5] oraz [4.6] przedstawiają kody programów, które są identyczne pod względem struktury i nazewnictwa, lecz inne pod względem ułożenia elementów. Analizując rysunki [4.5] oraz [4.6] możemy wnioskować, że nie jest to duży problem dla algorytmu wyszukiwania wspólnych poddrzew pod względem znalezienia ostatecznego rozwiązania, gdyż sama struktura drzew jest ułożona, jako inne dziecko, lecz pod względem złożoności, może być trudnym problemem.

```
1  package com.github.rkurcwald;
2
3  public class changeOrder {
4      int iterator = 1;
5      String name = "Name";
6
7      public static String main(String[] args)
8      {
9          name="Second "+name;
10         iterator+=2;
11         return name;
12     }
13
14 }
```

Listing 4.5: Kod źródłowy: changeOrder.java



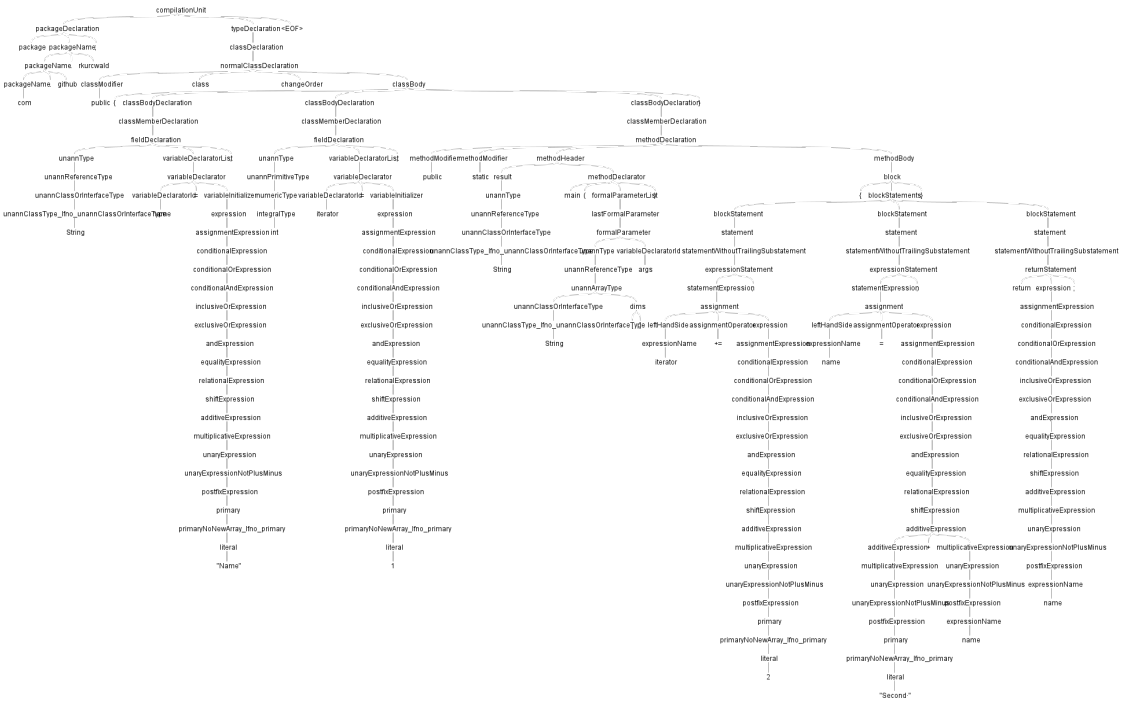
Rysunek 4.5: Drzewo parsowania wygenerowane dla changeOrder.java (Listing: 4.5)

```
1  package com.github.rkurcwald;
2
3  public class changeOrder {
4      String name = "Name";
5      int iterator = 1;
6
7      public static String main(String[] args)
8      {
9          iterator+=2;
10         name="Second "+name;
11         return name;
12     }
13
14 }
```

Listing 4.6: Kod źródłowy: changeOrder2.java (Przykład zmiany szyku)

4.1.4 Zastępowanie jednych elementów ich odpowiednikami

Problem wymagający od osoby zmieniającej oryginalny kod trochę więcej wysiłku niż wyżej wymienione techniki i bazuje na zmianie elementu na odpowiednik np. pętle for (Listing [4.7]) i foreach (Listing: [4.8]), które mają inne drzewa parsowania (Rysunki [4.7] i [4.8]). Pomysłem na taką technikę może być baza poddrzew dla elementów pokrewnych, w której przechowywane

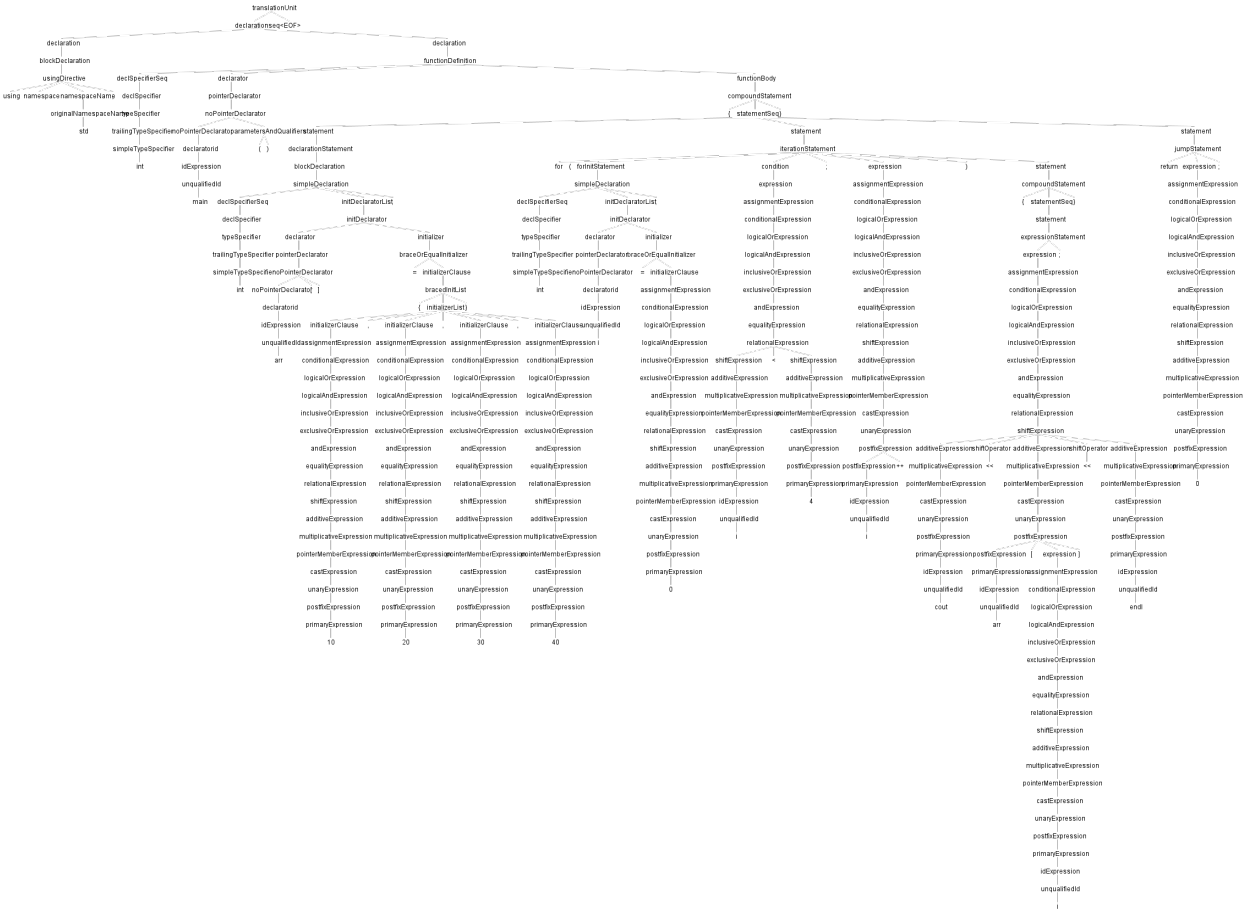


Rysunek 4.6: Drzewo parsowania wygenerowane dla changeOrder2.java (Listing: 4.6)

struktury byłyby dostępne w prosty sposób dla algorytmu sprawdzającego podobieństwo. Problemem mógłby być tu czas dostępu do bazy danych, czy nadmierna próba zamiany elementów strukturalnych celem ponownego sprawdzenia podobieństwa.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int arr[] = { 10, 20, 30, 40 };
6       for(int i=0;i<4;i++)
7       {
8            cout<<arr[i]<<endl;
9       }
10
11           return 0;
12 }
```

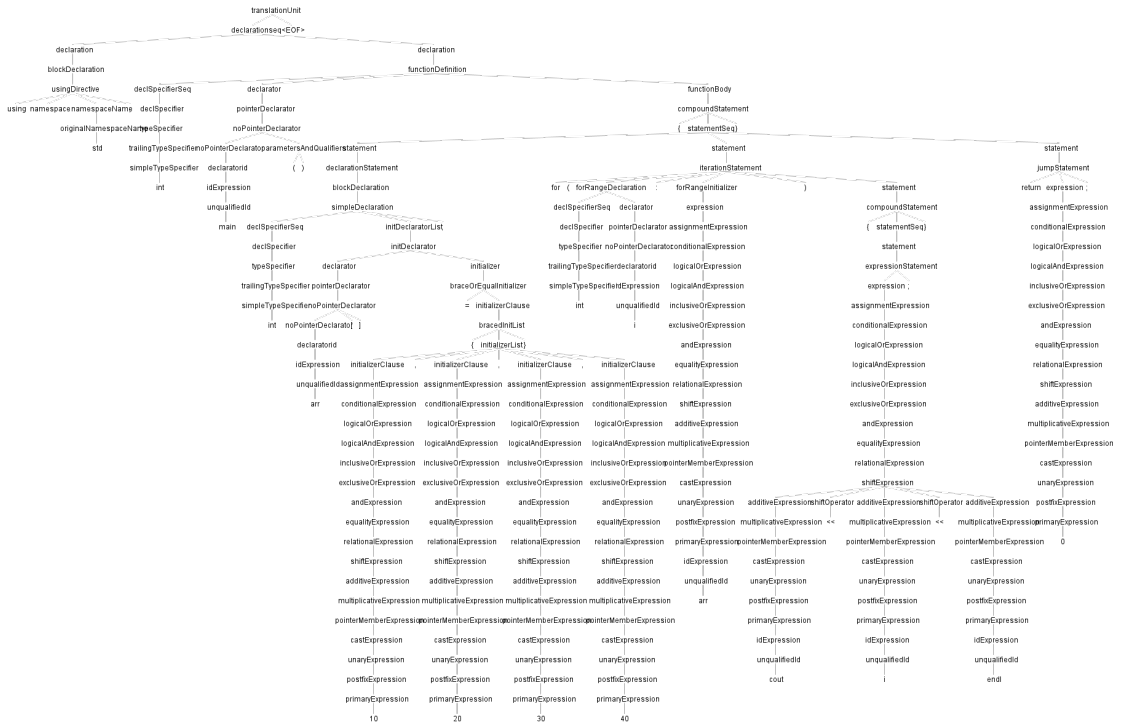
Listing 4.7: Kod źródłowy: for.cpp (Przykład użycia pętli for)



Rysunek 4.7: Drzewo parsowania wygenerowane dla for.cpp (Listing: 4.7)

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int arr[] = { 10, 20, 30, 40 };
6       for(int i: arr)
7       {
8            cout<<i<<endl;
9       }
10
11           return 0;
12 }
```

Listing 4.8: Kod źródłowy: foreach.cpp (Przykład użycia pętli foreach)



Rysunek 4.8: Drzewo parsowania wygenerowane dla foreach.cpp (Listing: 4.8)

4.1.5 Rozbijanie klas na mniejsze

Ten problem nie występuje często przy próbie modyfikowania kodu, gdyż wymaga zrozumienia przynajmniej podstawowych pojęć, jak dziedziczenie czy referencja do obiektu, a ponadto znacznie zwiększa się czas poświęcony na zmianę kodu. Cały problem polega na zmianie elementów i struktur zawartych w jednej klasie i rozdrobnieniu ich na kilka mniejszych klas. Proces znacząco wpływa na wygenerowanie drzewa parsowania, a finalnie na stopień podobieństwa.

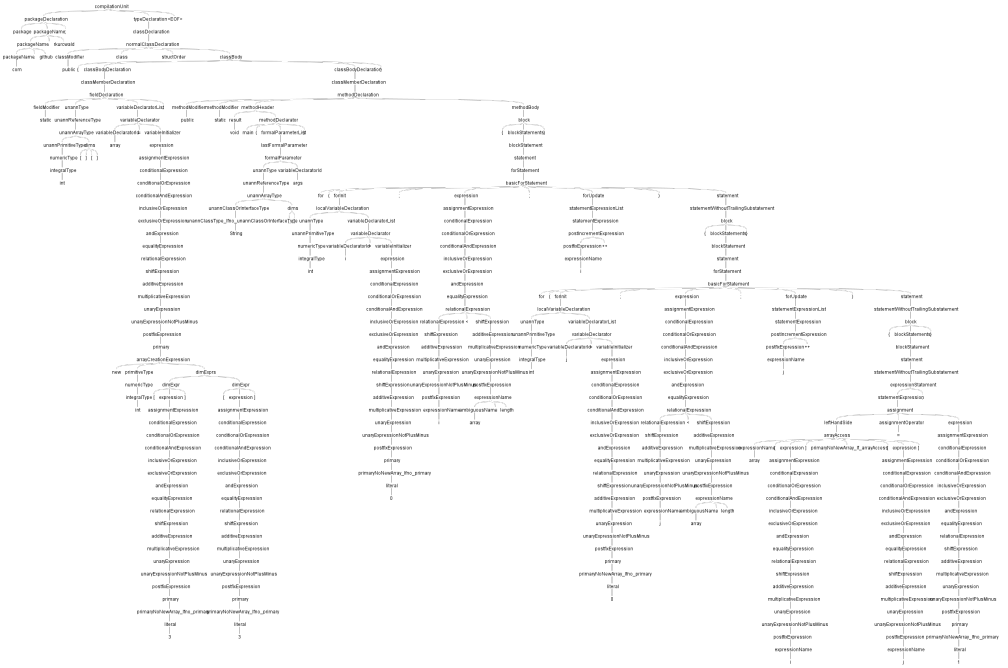
4.1.6 Strukturalne rozdrabnianie programu

Do takiego typu technik należy np. zagnieżdżona pętla for, która jest rozbijana na pojedyncze pętle for, ale druga z pętli jest wykonywana przez metodę. Przykład przedstawiono w Listingach [4.9] oraz [4.10]. Po analizie wygenerowanych drzew parsowania można stwierdzić, że jest to problem podobny do techniki zmiany szyku programu z tą różnicą, iż zamiast dalszej definicji struktury, otrzymano odwołanie do miejsca, gdzie następuje ta czynność. Możemy stwierdzić, że jest to swego rodzaju redundantność zmienionego szyku kodu.

```
1  package com.github.rkurcwald;
2
3  public class structOrder
4  {
5      static int[][][] array=new int[3][3];
6      public static void main(String[] args)
```

```
7       {
8           for(int i=0;i<array.length;i++)
9           {
10              for(int j=0;j<array.length;j++)
11              {
12                  array[i][j]=1;
13              }
14          }
15      }
16
17  }
```

Listing 4.9: Kod źródłowy: structOrder.java (Przykład pętli zagnieżdżonej)



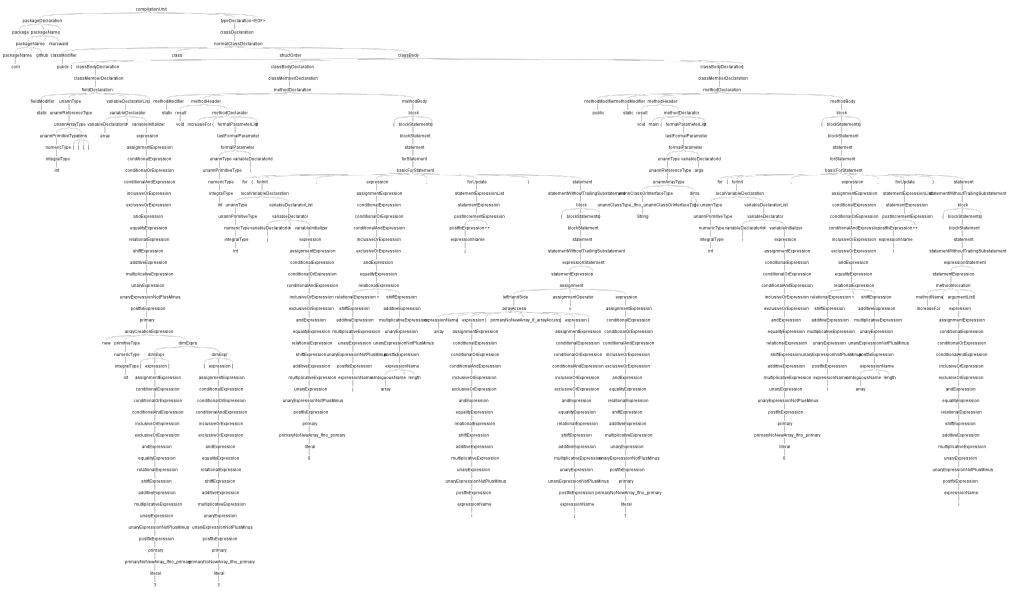
Rysunek 4.9: Drzewo parsowania wygenerowane dla structOrder.java (Listing: 4.9)

```java
package com.github.rkurcwald;

public class structOrder
{
    static int[][][] array=new int[3][3];
    static void increaseFor(int i)
    {
        for(int j=0;j<array.length;j++)
        {
            array[i][j]=1;
        }
    }
    public static void main(String[] args)
    {
        for(int i=0;i<array.length;i++)
        {
            increaseFor(i);
        }
    }

}
```

Listing 4.10: Kod źródłowy: structOrder2.java (Przykład pętli zewnętrznie zagnieżdżonej)



Rysunek 4.10: Drzewo parsowania wygenerowane dla structOrder2.java (Listing: 4.10)

Na koniec warto również wspomnieć, że zwykle osoba zmieniająca kod programu używa kilku, a w skrajnych przypadkach nawet wszystkich wyżej wymienionych metod (a także innych), dlatego idealnym byłoby stworzenie programu, który sprawdza wszystkie możliwe opcje, ale również posiada możliwość rozbudowy o kolejne moduły, które mogą być również uznane za

próbę zmiany kodu np. deklarowanie zmiennych, metod czy klas, które nigdy nie są wykorzystywane w kodzie czy tworzenie sztucznych metod, które mają na celu stworzenie innych poddrzew niewpływających na korelację między kodem wejściowym, a wynikiem działania tego kodu.

Rozdział 5

Implementacja oraz problemy z nią związane

Rozdział ten przedstawia w jaki sposób mechanizm sprawdzania został napisany, a także zapoznaje z problemami w jego implementacji. Ponadto sama implementacja nie została dołączona do pracy, gdyż była użyta tylko do przeprowadzenia badań. Cała struktura została przedstawiona w formie formalnego pseudokodu. [1, 2]

5.1 Założenia implementacyjne oraz definicja zależności

Próbując zniwelować problemy z rozdziału 4.1 musimy wyjść z pewnymi założeniami implementacyjno-użytkowymi:

- **Jeden plik na wejściu programu** - zakładamy, że użytkownik dostarcza tylko jeden kod źródłowy, który zawiera w sobie wszystkie klasy, metody, zmienne itp. Punkt jest ważny dla zmniejszenia problemu związanego z niekompletnym kodem programu na wejściu.
- Posiadamy kompletną i sprawdzoną gramatykę danego języka
- Badamy poprawność tylko poprawnie syntaktycznych programów

5.2 Schemat działania i badania wybranego algorytmu porównywania

Po spełnieniu wszystkich założeń z poprzedniego punktu, możemy przejść do definicji, bez której sam problem byłby niekompletny.
Stopień podobieństwa programów to wyliczona wartość liczbowa z porównywania drzew parsowania przy użyciu algorytmu wyszukiwania wspólnych kontekstów, gdzie główną metodą wyznaczania danych do przeszukiwania jest porównywanie kolejnych par wierzchołków dwóch

niezależnych programów. Wartość ta określana jest wzorem
$P_{XwY} = \frac{Liczba_{zaznaczonych\_jako\_odwiedzone\_wierzcholkow} - stala\_danego\_jezyka}{Liczba_{wszystkich\_wierzcholkow\_danego\_drzewa} - stala\_danego\_jezyka}$.

5.2.1 Wyprowadzenie stałych dla danych języków

Stałe dla danego języka są wyznaczone w procesie badania struktury kodu w następujący sposób:

1. Napisanie programu w danym języku programowania, który zawiera tylko funkcję "main"
2. Uruchomienie programu bazującego na ANTLR4 i stworzenie drzewa parsowania z gramatyki dla danego języka
3. Zapisanie wartości c dla danego języka licząc wszystkie wierzchołki tego drzewa.

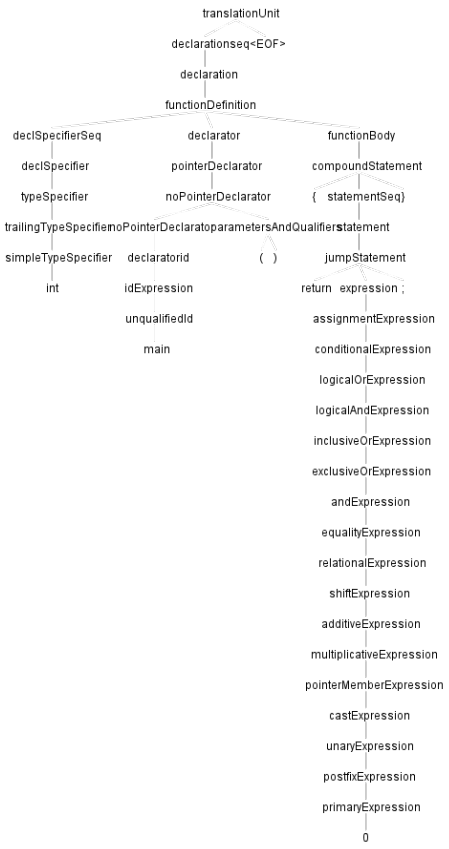
Zgodnie z założeniem 1. możemy przyjąć, że stałe potrzebne do wyznaczenia podobieństwa mają następujące wartości:

- C++ : stała $c = 49$ wierzchołków [Rysunek: 5.1]
- Java : stała $c = 38$ lub $c = 42$ wierzchołków [Rysunek: 5.2, 5.3]

Jeśli chodzi o stałą Javy, to w procesie badania zostały wyznaczone dwie stałe, gdzie ta wyższa odnosi się do słowa kluczowego *static*. W dalszej części przeprowadzonych badań używano natomiast stałej $c = 38$ (bez słowa *static*).

Używając zbadanych stałych dla danego języka możemy zauważyć pewne zależności, które mogą wpłynąć na finalny wynik procesu sprawdzania podobieństwa. Błędy o których mowa, wynikają ze sprawdzenia różnicy wynikającej pomiędzy wzorem z użytymi stałymi ($P_{XwY} = \frac{Liczba_{zaznaczonych\_jako\_odwiedzone\_wierzcholkow} - stala\_danego\_jezyka}{Liczba_{wszystkich\_wierzcholkow\_danego\_drzewa} - stala\_danego\_jezyka}$), a wzorem bez tych stałych ($P_{XwY} = \frac{Liczba_{zaznaczonych\_jako\_odwiedzone\_wierzcholkow\_drzewa\_X}}{Liczba_{wszystkich\_wierzcholkow\_drzewa\_Y}}$, przez co możemy przedstawić, jaka jest różnica między podobieństwem w zależności od długości kodu, a stałą językową c.
Poniżej przedstawiono błędy w punktach procentowych dla poszczególnego języka, które otrzymujemy w procesie porównywania
C++: [Rysunek: 5.4]

- błąd powyżej 20p.p dla drzew mniejszych, których liczba wierzchołków jest z przedziału [200;250].
- błąd między 10p.p, a 20p.p dla drzew, których liczba wierzchołków nie przekracza 500, z założeniem, że suma wierzchołków wspólnych kontekstów, jest mniejsza niż 140.

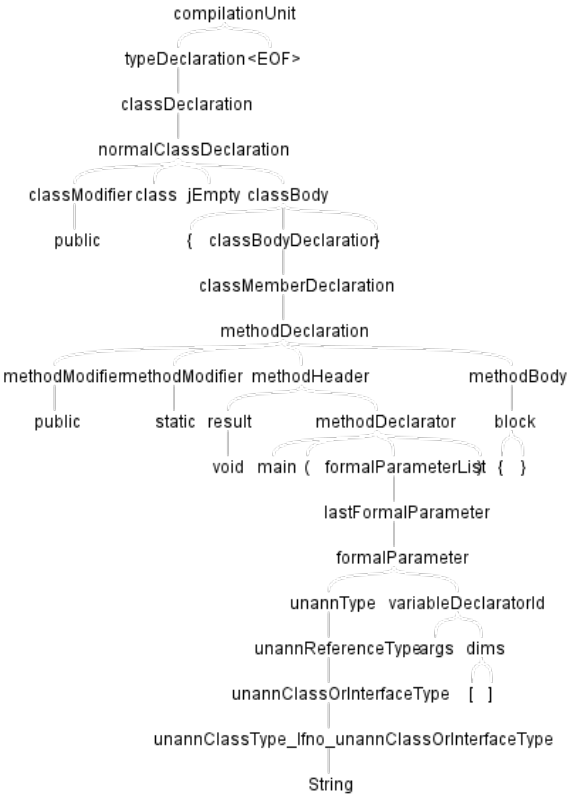


Rysunek 5.1: Drzewo parsowania wygenerowane celem znalezienia stałej 'c' dla języka C++

- błąd między 5p.p, a 10p.p dla drzew, z których liczba wierzchołków jest z przedziału [500;1225] wierzchołków z założeniem, że suma wierzchołków wspólnych kontekstów, nie jest mniejsza niż 50 (musi być większa niż stała języka c_cpp).
- błąd poniżej 5p.p dla drzew, których liczba wierzchołków jest większa niż 1225 wierzchołków z założeniem, że suma wierzchołków wspólnych kontekstów nie jest mniejsza niż stała językowa.

Java: [Rysunek: 5.5]

- błąd powyżej 20p.p dla drzew, których liczba wierzchołków jest mniejsza niż ok 170 wierzchołków.
- błąd między 10p.p, a 20p.p dla drzew, których liczba wierzchołków nie jest większa niż 370, z założeniem, że suma wierzchołków wspólnych kontekstów, jest mniejsza niż 110
- błąd między 5p.p, a 10p.p dla drzew, z których liczba wierzchołków jest z przedziału [370;760] z założeniem, że suma wierzchołków wspólnych kontekstów, nie jest mniejsza niż 39 (musi być większa niż stała języka c_java)



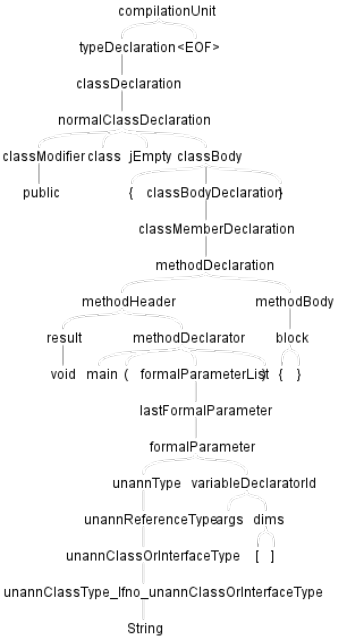
Rysunek 5.2: Drzewo parsowania wygenerowane celem znalezienia stałej 'c' dla języka Java

- błąd poniżej 5p.p dla drzew, których liczba wierzchołków jest większa niż 760 wierzchołków z założeniem, że suma wierzchołków wspólnych kontekstów nie jest mniejsza niż stała językowa

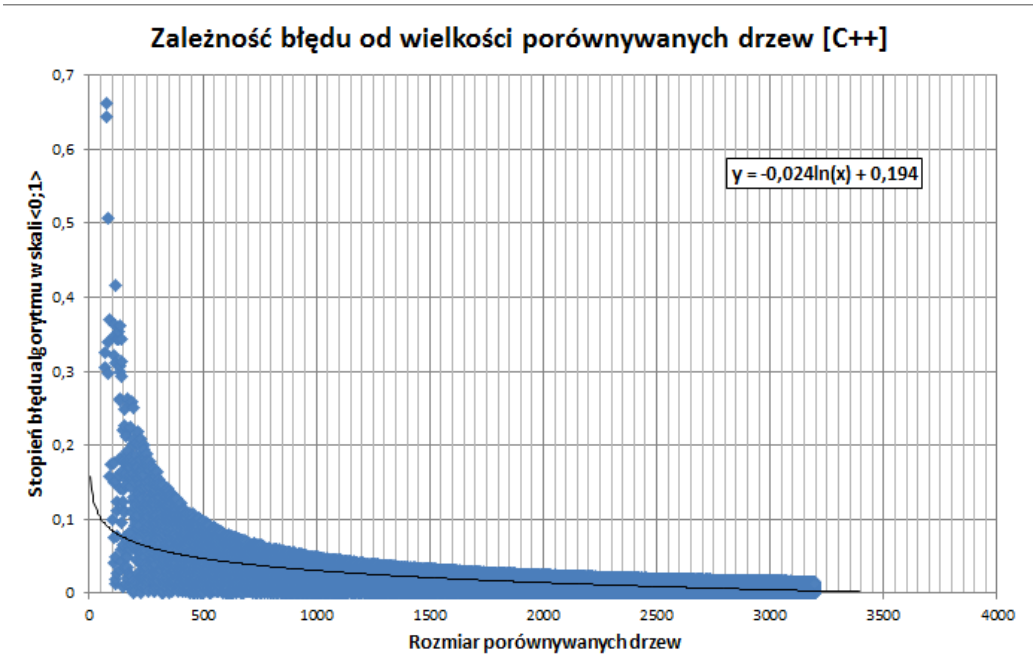
Podsumowując, końcowy wynik może różnić się od rzeczywistego w ściśle określony sposób np: porównując niewielkie programy typu for.cpp [Listing 4.7], uzyskujemy podobieństwo z błędem w granicach 10-20p.p, gdyż tego typu programy mają drzewa rozmiaru większego niż 200, ale mniejszego niż 500 (ok 350 wierzchołków). Natomiast jeśli weźmiemy bardziej złożony program np. Heapsort.cpp (Załączniki: Listing Heapsort.cpp, Rysunek: Heapsort.png) uzyskujemy podobieństwo z błędem w granicach 0,2%, gdyż drzewo wyprodukowane przez ANTLR ma wielkość ok. 12500 wierzchołków.

5.3 Opis programu

Proces implementacji narzędzia był bardzo złożony i niejednokrotnie zmieniał się wraz z napotkanymi błędami, a także ze zmianą koncepcji projektu. Początkowo do czystego projektu został

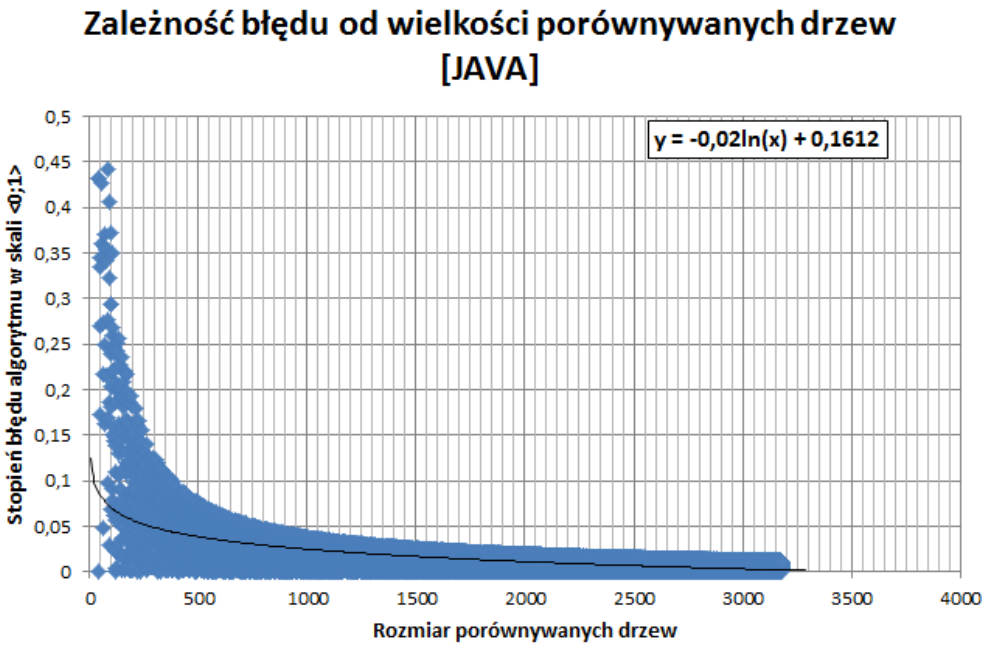


Rysunek 5.3: Drzewo parsowania wygenerowane celem znalezienia stałej 'c' dla języka Java (bez słowa kluczowego *static*)



Rysunek 5.4: Zależność błędu od wielkości drzew parsowania przy użyciu stałej 'c' dla języka C++

zaimportowany ANTLR, dzięki któremu można było wygenerować odpowiednie klasy (Parser, Listener, ParseTree itd) do danych języków. Następnym krokiem była integracja elementów gramatycznych pochodzących z analizatora składni do środowiska programistycznego. W tym



Rysunek 5.5: Zależność błędu od wielkości drzew parsowania przy użyciu stałej 'c' dla języka Java

momencie mechanizm mógł zacząć pracować nad wygenerowaniem drzew parsowania, które były wprowadzone do algorytmu porównywania opartego na wspólnych kontekstach. Na koniec dostawaliśmy zmiennoprzecinkowy wynik podobieństwa wprowadzonych kodów z dokładnością do czterech miejsc znaczących. [Pseudokod 5]

5.4 Problemy implementacyjne

Głównym problemem związanym z implementacyjną częścią pracy były funkcje związane z ANTLR'em i dopasowaniem ich do problemu podobieństwa. Dodatkowym problemem był początkowy czas zapoznania się z architekturą generatora parserów i proste błędy wynikające z nieznajomości składni lub przeoczeń np. przy próbie generowania drzewa parsowania dla języka Java, zamiast tworzyć nowy obiekt typu ParseTree, próbowano przypisać wartość tablicy dwuwymiarowej [Parser,Lexer]. Początkowo dodatkowym problemem była również źle zaimplementowana obsługa wyjątków przez autora, przez co niekiedy znalezienie problemu trwało długi czas.

Pseudokod 5: Wyznaczenie stopni podobieństwa dwóch programów

1 **Założenia:**

- **Na wejściu mamy 2 pliki. Każdy z nich to osobny program do porównania**
- **Każdy z plików posiada funkcję main**

2 SimilarityCheck(IOFile args[] (.cpp/.java)):
3 ParseTree PT0=generateParseTree(args[0]) ▷ Generuj drzewo używając ANTLR dla danego języka
4 ParseTree PT1=generateParseTree(args[1]) ▷ Generuj drzewo używając ANTLR dla danego języka
5 setSigma0=biggestSubtree(PT0,PT1)
6 setSigma1=biggestSubtree(PT1,PT0)
7 nodesCountPT0=PT0.getNodesCount()
8 nodesCountPT1=PT1.getNodesCount()
9 $P_{PT0wPT1} = \frac{(setSigma0 - c)}{(nodesCountPT0 - c)}$ ▷ Podobieństwo programu 0 w 1
10 $P_{PT1wPT0} = \frac{(setSigma1 - c)}{(nodesCountPT1 - c)}$ ▷ Podobieństwo programu 1 w 0
11 $P_{sr} = \frac{P_{PT0wPT1} + P_{PT1wPT0}}{2}$ ▷ Podobieństwo uśrednione
12 return $P_{PT0wPT1}, P_{PT1wPT0}, P_{sr}$

Rozdział 6

Badania i wyniki

Rozdział przedstawia wyniki przeprowadzonych eksperymentów na realnych danych, a także na prostych programach kilkunastolinijkowych, które były różnymi programami rozwiązującymi ten sam problem. Stosując metody podane w rozdziale 4 można było zaburzyć podobieństwo programów nawet do 10 p.p przy zastępowaniu jednych elementów ich odpowiednikami, a przy rozbijaniu klas na mniejsze, ta wartość zwykle oscylowała w granicach 15 p.p.

6.1 Badania programów studenckich

Przy badaniu realnych programów wychodzimy z założenia, że każdy kod będzie miał usunięte komunikaty wypisywane go konsoli oprócz komunikacji wynikowej.

6.1.1 Realny przykład

Przykładem badanego kodu jest rozwiązanie jednego z pierwszych zadań z przedmiotu "Algorytmy i struktury danych", osób A [Listing 6.1] oraz B [Listing 6.2] w technologii C++. Zadanie polegało na wpisaniu w tablicę rzędu kilkuset tysięcy losowej liczby wartości maksymalnej $2^{63}-1$ z przedziału <0;10000) i posortowaniu elementów używając sortowania bąbelkowego. Wygenerowane drzewa parsowania były zbyt duże by umieścić je w pracy, dlatego zostały dołączone jako załączniki (Załącznik: bubbleSortA.png, bubbleSortB.png).

```
1  #include <iostream>
2  #include <conio.h>
3  #include <ctime>
4  #include <cstdlib>
5
6
7  long int a[100000000];
8  long long t;
9  using namespace std;
10
11 int main()
```

```
12  {
13          srand(time(NULL));
14          int p, j, i, n, wybor;
15          double c;
16  //      printf("Wielkosc tablicy\n");
17          cin >> n;
18          clock_t start = clock();
19
20                  for (int i = 1; i <= n; i++)
21                  {
22                          a[i] = rand();
23                  }
24
25
26
27          for (j = 2; j <= n; j++)
28          {
29                  p = a[j];
30                  i = j - 1;
31                  while ((i > 0) && (a[i]> p))
32                  {
33                          a[i + 1] = a[i];
34                          i = i - 1;
35                          a[i + 1] = p;
36                          t++;
37                  }
38          }
39          printf("Czas wykonywania: %lu ms\n",
               ↪ clock() - start);
40          cout << "\n wykonanych porownan: " << t;
41          c = double(t) / (double(n)*double(n));
42          cout << " iloraz" << c;
43
44          _getch();
45  }
```

Listing 6.1: Kod źródłowy: bubbleSortA.cpp (Sortowanie bąbelkowe osoby A)

```
1  #include<iostream>
2  #include<time.h>
3  using namespace std;
4  void wypisz(int tablica[], long long n,int t)
5  {
6          cout << "Tablica posortowana:\n";
7          for (int i = 0; i < n; i++)
8          {
9         cout << "tablica[" << i+1 << "] = " <<
             ↪ tablica[i] << endl;
```

```
10          }
11          float c = static_cast<float>(t) / (n*n);
12          cout << "liczba porownan wynosi = " << t
                ↪  <<", a C wynosi: "<<c<< endl;
13  }
14
15  void sort(int tablica[], long long n)
16  {
17          int j, p,t=0;
18
19          for (int i=1;i<n; ++i)
20                  {
21                    j=i;
22                          while (tablica[j - 1] >
                               ↪ tablica[j] && j>0)
23                          {
24                                  p = tablica[j];
25                                  tablica[j] =
                                       ↪ tablica[j
                                       ↪ - 1];
26                                  tablica[j - 1] =
                                       ↪  p;
27                                  j--;
28                                  t++;
29                          }
30                          t++;
31                  }
32          wypisz(tablica, n,t);
33  }
34  int main()
35  {
36          long long n;
37  //      cout << "Podaj ilosc liczb do
        ↪ posortowania: ";
38          cin >> n;
39          int *tablica = new int[n];
40          srand(time(NULL));
41          for (int i = 0; i < n; i++)
42          {
43                  tablica[i] = rand() % 10000;
44          }
45          sort(tablica, n);
46          system("PAUSE");
47          return 0;
48  }
```

Listing 6.2: Kod źródłowy: bubbleSortB.cpp (Sortowanie bąbelkowe osoby B)

Algorytm wyszukiwania największych wspólnych poddrzew wyznaczył stopień podobieństwa $P_{BwA} = 0,79375$ oraz $P_{AwB} = 0,6125$ co sugeruje, że program A jest dłuższy i użyto w nim innych metod czy struktur. Dodatkowo podobieństwo uśrednione wynosi $P_{sr} = 0,703125$ [Rysunek: 6.1] co może sugerować, że programy zostały napisane samodzielnie przy użyciu tego samego pseudokodu.

```
L1_A -- saved
1511 Relative: 1462
L1_B -- saved
1938 Relative: 1889
P_{BwA}=0,79375
P_{AwB}=0,6125
P_{sr}=0,703125
EOF
```

Rysunek 6.1: Konsola z wynikami

6.1.2 Schemat uśrednienia wyników podobieństwa

Po przebadaniu realnych programów z przedmiotu "Algorytmy i struktury danych" możemy stwierdzić, że wszystkie programy bazujące na tych samych pseudokodach pokazywanych przez prowadzącego mają wysoki stopień podobieństwa, który zwykle oscyluje w granicach 60-75% z błędem w granicach 5-10p.p przy dwóch pierwszych listach. Sytuacja ma się inaczej od listy 3, gdzie uśrednione wyniki stopnia podobieństwa wahały się w granicach 50-80% z błędem w granicach 2-3p.p, co może sugerować, że niektóre programy zostały napisane używając innych pseudokodów (schematów), jednakże jest też grupa osób, które swoje programy pisały niesamodzielnie, a podobieństwa między niektórymi parami wynosiły ok 90% 1-2p.p.
Warto również wspomnieć, że przy niektórych porównaniach występowały duże różnice w ilości kodu, gdyż niektóre osoby rozszerzały swój program o kolejne listy, a inne oddawały każdy program osobno, dlatego każdy program musiał zostać podzielony na osobną listę celem porównania i uśrednienia wyników podobieństwa dla danego problemu.

6.1.3 Zbadanie wpływu zdefiniowanych problemów przy porównywaniu, na końcowe wyniki podobieństwa

Badając wpływ konkretnych problemów zdefiniowanych w rozdziale 4, na podstawie programu bubbleSortA.cpp [6.1] i jego modyfikacji [Lising: 6.3, 6.4, 6.5][Załączniki: *bubbleSortA_nameMod.png*,

bubbleSortA_redundant.png, *bubbleSortA_swap.png*], możemy stwierdzić, że:

- Uśredniony stopień podobieństwa programu *bubbleSortA.cpp* do *bubbleSortA_nameMod.cpp* wynosi 98,72% z błędem ok 1p.p.
- Uśredniony stopień podobieństwa programu *bubbleSortA.cpp* do *bubbleSortA_swap.cpp* wynosi 99,48% z błędem ok 1p.p.
- Uśredniony stopień podobieństwa programu *bubbleSortA.cpp* do *bubbleSortA_redundant.cpp* wynosi 95,83% z błędem ok 1p.p.

Możemy stwierdzić, że dzięki heurystycznemu algorytmowi przeszukiwania kontekstów, stopień podobieństwa przy drobnych modyfikacjach, które zarazem są najbardziej popularnymi metodami wykorzystywanymi do zmiany wyglądu kodu, jest bardzo wysoki co sugeruje, że praca z dużym prawdopodobieństwem jest kopią pierwszej z wymienionych.

```
1  #include <iostream>
2  #include <conio.h>
3  #include <ctime>
4  #include <cstdlib>
5
6
7  long int tab[100000000];
8  long long counter;
9  using namespace std;
10
11 int main()
12 {
13         srand(time(NULL));
14         int tmp, sndIndex, fstIndex, input,
               ↪ choose;
15         double value;
16 //      printf("Wielkosc tablicy\input");
17         cin >> input;
18         clock_t start = clock();
19
20                 for (int fstIndex = 1; fstIndex
                       ↪ <= input; fstIndex++)
21                 {
22                         tab[fstIndex] = rand();
23                 }
24
25
26
27         for (sndIndex = 2; sndIndex <= input;
               ↪ sndIndex++)
28         {
```

```
29                  tmp = tab[sndIndex];
30                  fstIndex = sndIndex - 1;
31                  while ((fstIndex > 0) && (tab[
                        ↪ fstIndex]> tmp))
32                  {
33                          tab[fstIndex + 1] = tab[
                                ↪ fstIndex];
34                          fstIndex = fstIndex - 1;
35                          tab[fstIndex + 1] = tmp;
36                          counter++;
37                  }
38          }
39          printf("Czas wykonywania: %lu ms\input",
                ↪  clock() - start);
40          cout << "\input wykonanych porownan: "
                ↪ << counter;
41          value = double(counter) / (double(input)
                ↪ *double(input));
42          cout << " iloraz" << value;
43
44          _getch();
45  }
```

Listing 6.3: Kod źródłowy: bubbleSortAnameMod.cpp (Sortowanie bąbelkowe osoby A ze zmodyfikowanymi nazwami)

```
1  #include <iostream>
2  #include <conio.h>
3  #include <ctime>
4  #include <cstdlib>
5
6
7  long int a[100000000];
8  long long t;
9  using namespace std;
10
11 int main()
12 {
13          srand(time(NULL));
14          int p, j, i, n, wybor;
15          double c;
16          cin >> n;
17          clock_t start = clock();
18          int redu1=1000;
19
20          if(redu1==500)
21          {
22                  cout<<"500"<<endl;
```

```
23          }
24                  for (int i = 1; i <= n; i++)
25                  {
26                          a[i] = rand();
27                  }
28
29
30
31          for (j = 2; j <= n; j++)
32          {
33                  p = a[j];
34                  i = j - 1;
35                  while ((i > 0) && (a[i]> p))
36                  {
37                          a[i + 1] = a[i];
38                          i = i - 1;
39                          a[i + 1] = p;
40                          t++;
41                          t=t-1;
42                          t=t+1;
43                  }
44          }
45          printf("Czas wykonywania: %lu ms\n",
                ↪ clock() - start);
46          cout << "\n wykonanych porownan: " << t;
47          c = double(t) / (double(n)*double(n));
48          cout << " iloraz" << c;
49
50          _getch();
51  }
```

Listing 6.4: Kod źródłowy: bubbleSortAredundant.cpp (Sortowanie bąbelkowe osoby A z dopisanym kodem o dużej niezależności)

```
1  #include <iostream>
2  #include <conio.h>
3  #include <ctime>
4  #include <cstdlib>
5
6  long long t;
7  long int a[100000000];
8  double c;
9  int p, j, i, n, wybor;
10 using namespace std;
11
12 int main()
13 {
14          srand(time(NULL));
```

```
15
16          cin >> n;
17          clock_t start = clock();
18
19                  for (int i = 1; i <= n; i++)
20                  {
21                          a[i] = rand();
22                  }
23
24
25
26          for (j = 2; j <= n; j++)
27          {
28                  p = a[j];
29                  i = j - 1;
30                  while ((i > 0) && (a[i]> p))
31                  {
32                          a[i + 1] = a[i];
33                          i = i - 1;
34                          a[i + 1] = p;
35                          t++;
36                  }
37          }
38          c = double(t) / (double(n)*double(n));
39          cout << " iloraz" << c;
40          printf("Czas wykonywania: %lu ms\n",
               ↪ clock() - start);
41          cout << "\n wykonanych porownan: " << t;
42
43
44          _getch();
45  }
```

Listing 6.5: Kod źródłowy: bubbleSortAswap.cpp (Sortowanie bąbelkowe osoby A z zamienioną kolejnością elementów)

6.2 Wnioski

W badaniach użyto tylko implementacji algorytmu wyszukiwania wspólnych kontekstów [Pseudokod: 1, 2], więc nie mamy pełnego porównania wyników z heurystyką przedstawioną w pseudokodach: [3, 4]. Jednakże możemy stwierdzić, że przy zadaniach akademickich z podanym przez prowadzącego pseudokodem programu, podobieństwo jest wysokie, gdyż program oparty jest o ten sam schemat, co może sugerować, że prace wykonane samodzielnie niekoniecznie takimi są. Inaczej sytuacja wygląda, jeśli w problemie nie ma pokazanego pseudokodu, to wynik działania algorytmu największych wspólnych poddrzew jest bardziej miarodajny. W rozdzia-

le 5.2 pokazano, że błąd dotyczący ostatecznych wyników jest zmienny i zależy od wielkości wygenerowanego drzewa parsowania z kodu programu.

Rozdział 7

Podsumowanie

Celem pracy było stworzenie schematu porównywania programów jedno-plikowych, który wskaże stopień podobieństwa plików wejściowych, a także wyznaczenia stopnia podobieństwa, w jakim studenci podczas pisania programów wzorują się z pseudokodów czy gotowych rozwiązań tego problemu. Przedstawione w pracy heurystyki pozwoliły na sprawdzenie stopni podobieństwa, a nawet wyznaczenie uśrednionych wartości podobieństwa dla danego problemu. Możemy stwierdzić, iż cel został osiągnięty połowicznie, jeśli chodzi o implementacje, w której zabrakło porównania wyników obu heurystyk, gdyż druga ze zdefiniowanych, nie została zaimplementowana. Miejscem do poprawy, a zarazem dalszą możliwością rozwoju, może być opracowanie innego schematu opartego o pętle, a nie instrukcje *goto*, by dokładniej oszacować złożoność obliczeniową.

Pliki źródłowe badanych programów [CPP]

Wszystkie pliki źródłowe są dodane w lokalizacji: ./file/antlr/examples/cpp/

Pliki źródłowe badanych programów [Java]

Wszystkie pliki źródłowe są dodane w lokalizacji: ./file/antlr/examples/java8/

Wygenerowane obrazy drzew parsowania [CPP]

Wszystkie wygenerowane drzewa parsowania znajdują się w lokalizacji: ./file/target/genImages/CPP/

Wygenerowane obrazy drzew parsowania [Java]

Wszystkie wygenerowane drzewa parsowania znajdują się w lokalizacji: ./file/target/genImages/Java/

Wygenerowane obrazy drzew parsowania [Java]

Wszystkie wygenerowane drzewa parsowania znajdują się w lokalizacji: ./file/target/genImages/Java/

Wygenerowane drzewa parsowania w formie tekstowej [CPP]

Wszystkie wygenerowane drzewa parsowania w formie tekstowej znajdują się w lokalizacji: ./file/target/genTxt/CPP/

Wygenerowane drzewa parsowania w formie tekstowej [Java]

Wszystkie wygenerowane drzewa parsowania w formie tekstowej znajdują się w lokalizacji: ./file/target/genTxt/Java/

Bibliografia

[1] Ira D. Baxter; Andrew Yahin; Keonardo Moura; Marcelo Sant' Anna; Lorraine Bier. *Clone Detection Using Abstract Syntax Trees*. Odwiedzono: 14.06.2021. 1998.

[2] MacBride Fraser. "Relations". In: *The Stanford Encyclopedia of Philosophy*. Odwiedzono: 01.06.2021. Metaphysics Research Lab, Stanford University, 2020. URL: https://plato.stanford.edu/archives/win2020/entries/relations/.

[3] Terence Parr i inni. *ANTRL v4*. P1.0. Odwiedzono: 31.05.2021. 2013. URL: https://github.com/antlr/antlr4/blob/master/doc/.

[4] Terence Parr i inni. *ANTRL v4 Grammars*. Odwiedzono: 31.05.2021. 2012-2021. URL: https://github.com/antlr/grammars-v4.

[5] Alexander S. Kechris. *Classical Descriptive Set Theory*. Springer New York, 1995. DOI: 10.1007/978-1-4612-4190-4. URL: https://doi.org/10.1007%2F978-1-4612-4190-4.

[6] Rainer Koschke and Saman Bazrafshan. "Software-Clone Rates in Open-Source Programs Written in C or C++". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 3. Odwiedzono: 14.06.2021. 2016, pp. 1–7. DOI: 10.1109/SANER.2016.28.

[7] Oracle. *Java SE Development Kit 8 Documentation*. 1.0. Odwiedzono: 23.04.2021. Redwood City, CA 94065, U.S.A, 2014. URL: https://www.oracle.com/pl/java/technologies/javase-jdk8-doc-downloads.html.

[8] Terence Parr. *ANTRL v4 - About us*. Odwiedzono: 31.05.2021. 2021. URL: https://www.antlr.org/about.html.

[9] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012. URL: https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/.

Spis listingów

Spis rysunków

Spis pseudokodów

Załączniki