Akademia Górniczo-Hutnicza im. Stanisława Staszica

w Krakowie

System wykrywania podobieństw kodów źródłowych w projektach studenckich

Praca dyplomowa

Jarosław Szczęśniak

Promotor: dr inż. Darin Nikolow

26 września 2012

Spis treści

1 Wstęp

2 Cel i zakres prac

Celem pracy jest zbudowanie systemu, który będzie wspomagał wykrywanie podobieństw w danych kodach źródłowych. Docelowym zagadnieniem z jakim ma zmierzyć się system jest porównywanie kodów źródłowych napisanych w języku Asembler.

Osiągnięcie celu głównego wymaga zrealizowania szeregu celów cząstkowych:

- zapoznanie się z dostępnymi metodami porównywania tekstu
- zdefiniowanie języka wewnętrznego:
  - symboli leksykalnych
  - gramatyki języka
  - akcji semantycznych
- implementacja co najmniej jednej metody porównywania
- dostarczenie aplikacji analizującej kod źródłowy
- dostarczenie środowiska do porównywania kodów źródłowych
- dostarczenie przykładowych programów w języku wewnętrznym

Pierwszym etapem całego mechanizmu powinien być etap preprocessingu, czyli odpowiedniego przefiltrowania plików źródłowych, celem usunięcia zbędnych białych znaków oraz komentarzy.

Następnym krokiem jest proces analizy leksykalnej, którego zadaniem jest wyłonienie odpowiednich symboli leksykalnych, które zostaną przekazane do analizatora składniowego.

Kolejnym krokiem jest parsowanie, które odbywa się za pomocą analizatora składniowego. Sprawdza on, czy przekazane symbole wejściowe są zgodne z gramatyką języka oraz buduje z nich tzw. drzewo składniowe, które będzie wykorzystywane podczas porównywania dwóch kodów źródłowych.

Istnieje kilka skutecznych metod porównywania tekstu, poczynając od najprostszej i zarazem najmniej efektywnej metody brute-force, poprzez szybsze funkcje wykorzystujące zróżnicowane metody (np. funkcje hashujące), do algorytmów wykorzystujących deterministyczne automaty skończone.

Etap analizy można podzielić na 3 elementarne części:

- analizę leksykalną – wczytywanie znaków i grupowanie ich w większe symbole
- analizę składniową – sprawdzenie poprawności z gramatyką, stworzenie drzewa składniowego
- analizę semantyczną – analizę kodu na podstawie drzewa składniowego, zapewnienie odpowiednich warunków (zgodność typów, wywoływanie odpowiednich funkcji, sprawdzanie zasięgów)

Założenia co do funkcjonalności języka wewnętrznego:

- obsługa mechanizmu tworzenia zmiennych
- obsługa pętli
- obsługa instrukcji warunkowych
- sprawdzanie wywoływanych funkcji
- mechanizmu obsługi wywoływanych funkcji (ilość przekazywanych parametrów, zwykłe oraz rekurencyjne wywoływanie funkcji)

Wynikiem tego etapu będzie łatwa do porównania lista symboli w postaci <function, value>, <variable, value>.

3 Teoria kompilacji

Kompilator to program, który tłumaczy kod źródłowy na kod wynikowy. Składa się on z dwóch głównych etapów - analizy i syntezy.

Etap analizy składa się z trzech etapów pośrednich - analizy leksykalnej, składniowej i semantycznej. Polega on na rozłożeniu kodu źródłowego na czynniki składowe oraz na budowie jego reprezentacji pośredniej.

Ten właśnie etap wydaje się być odpowiednim procesem, dzięki któremu można będzie wykonać porównanie dwóch kodów źródłowych na odpowiednio niskim poziomie, aby uniezależnić się od wykorzystywanego nazewnictwa lub organizacji i kolejności instrukcji w kodach źródłowych.

Język formalny to podzbiór zbioru wszystkich wyrazów nad skończonym alfabetem. Język formalny jest podstawowym pojęciem w informatyce, logice matematycznej i językoznawstwie. Aby zdefiniować język formalny musimy najpierw zdefiniować jego alfabet, składający się z symboli. Ciągi symboli nazywamy napisami, a dowolny zbiór tych ciągów to język formalny.

3.1 Symbole

Symbole leksykalne to abstrakcyjne ciągi znaków, które definiowane są podstawie określonych przez analizator reguł i przekazywane do dalszych etapów analizy. Reguły, według których są one definiowane, budowane są najczęściej z wyrażeń regularnych. Z symbolami leksykalnymi skojarzona jest też wartość leksykalna.

<przyklad>
sum = 3 + 2 ;
symbol | wartość
identyfikator | sum
operator przypisania | =
liczba | 3
operator dodawania | +
liczba | 2

koniec operacji | ;
</przyklad>

Symbole, których używany najczęściej to litery lub cyfry. Alfabet jest to niepusty zbiór, składający się ze skończonej liczby symboli. Przykładem alfabetu, może być alfabet języka polskiego lub alfabet Morse'a.

Napis jest to skończony ciąg symboli, które należą do podanego alfabetu. Długość napisu oznaczamy jako |n| i jest to ilość symboli w napisie „n", przykładowo długość |tekst| wynosi 5.

Prefiksem napisu nazywamy pewną liczbę symboli rozpoczynających napis. Sufiksem nazywamy pewną liczbę symboli kończących napis.

Język jest dowolnym zbiorem napisów ustalonym nad pewnym określonym alfabetem. Językiem może być zbiór pusty, zbiór zawierający pusty napis lub pewien podzbiór ze zbioru wszystkich łańcuchów nad określonym alfabetem.

3.2 Wyrażenia regularne

3.3 Gramatyka

To, co definiuje leksemy, to zestaw reguł określonych na podstawie specyfikacji języka programowania, czyli gramatyki.

Obecnie najczęściej stosuje się wyrażenia regularne do zdefiniowania reguł. Określają one zestaw możliwych sekwencji, które są użyte do zbudowania konkretnych leksemów.

W niektórych językach programowania kod dzieli się na bloki za pomocą par symboli (np. „{'' i „}"), które razem z wszelkimi białymi znakami również opisane są za pomocą leksemów, lecz nie są brane pod uwagę podczas dalszych etapów analizy.

3.4 Analiza leksykalna

Analiza leksykalna służy do dzielenia strumienia znaków wejściowych na grupy (symbole), do których dopasowywane są pewne sekwencje (leksemy) na podstawie reguł

(wzorców), w celu przekazania ich do dalszego etapu analizy. Leksem to abstrakcyjna jednostka leksykalna, która może obejmować słowa kluczowe, identyfikatory, etykiety, operatory, separatory, komentarze lub inne symbole specjalne.

3.4.1 Skaner

Skaner, czyli podstawowa część analizatora leksykalnego, zajmuje się głównym etapem analizy. Posiada on informacje na temat sekwencji znaków, które mogą być wykryte i zapisane jako symbole.

W wielu przypadkach pierwszy znak, który nie jest znakiem białym może posłużyć do dedukcji, z jakiego rodzaju symbolem mamy do czynienia (longest-match rule). W przypadku bardziej skomplikowanych języków potrzebna jest możliwość cofania się do wcześniej wczytanych znaków.

Skaner wczytuje kolejne znaki ze strumienia wejściowego i klasyfikuje je według reguł w symbole leksykalne, które następnie przekazywane są do analizy składniowej. Jako jednostka rozpoczynająca analizę i wczytująca tekst źródłowy może służyć również jako filtr, który pomija pewne elementy, takie jak np. komentarze, białe znaki lub znaki nowego wiersza.

Możliwe jest i czasem konieczne napisanie własnego analizatora, aczkolwiek jest to zwykle zbyt skomplikowane, czasochłonne i nieopłacalne. Z tego względu analizatory, zwane lekserami, zwykle generowane są za pomocą specjalnych narzędzi, które przyjmują na wejściu wyrażenia regularne, które opisuję wymagane symbole leksykalne.

3.5 Analiza składniowa

Analiza składniowa to proces analizy tekstu, złożonego z symboli, którego celem jest wyklarowanie jego struktury gramatycznej na podstawie zdefiniowanej gramatyki.

3.5.1 Parser

Jest to komponent, który wykonując analizę składniową, sprawdza jej poprawność i tworzy pewną strukturę danych (często tzw. drzewo składniowe), składającą się

z symboli wejściowych. Korzysta z analizatora leksykalnego, który zaopatruje go w zestaw symboli leksykalnych zbudowanych ze strumienia wejściowego. Innymi słowy parser analizuje kod źródłowy i tworzy jego wewnętrzną reprezentację.

3.5.2 Typy parserów

Zadaniem parsera jest określenie czy i w jaki sposób symbole leksykalne mogą być przekształcone na symbole gramatyczne. Istnieją dwa sposoby, dzięki którym osiągniemy rządany efekt:

- parsowanie zstępujące (top-down) – polega na znalezieniu symboli wysuniętych najbardziej na lewo przeszukując drzewo składniowe z góry na dół.
- parsowanie wstępujące (bottom-up) – polega na sprawdzeniu począwszy od słowa wejściowego i próbie redukcji do symbolu startowego, analizę zaczynamy od liści drzewa posuwając się wkierunku korzenia.

3.6 Opis procesu

4 Metody porównywania

Każda z opisanych metod posiada szereg cech ją charakteryzujących. Są to:

- faza preprocesingu - przygotowania danych wejściowych do analizy
- ilość potrzebnej pamięci dodatkowej
- złożoność czasowa - ilość wykonywanych operacji względem ilości danych wejściowych

4.1 Algorytm brute-force

Metoda brute – force, w teorii, jest najskuteczniejszą metodą, ponieważ sukcesywnie sprawdza wszystkie możliwe kombinacje w poszukiwaniu rozwiązania problemu. W praktyce jednak, algorytmy oparte na tej metodzie są niezwykle nieoptymalne, ze względu na czas wykonywania, przez co są rzadko stosowane.

Algorytm polega na sprawdzeniu wszystkich pozycji w tekscie pomiędzy znakiem 0 i $n-m$, gdzie m to długość poszukiwanego wzorca, a n to długość tekstu. Algorytm weryfikuje, czy wzorzec zaczyna się na danej pozycji. Jeśli tak to pozycja elementu w tekscie zapisywana jest do bufora. Następnie przesuwa wskaźnik w prawo po każdej próbie. Jeśli na kolejnych pozycjach znajdują się wszystkie kolejne elementy z wzorca to do tablicy wynikowej przepisywany jest zawartość bufora. Algorytm może być wykonywany w dowolnym kierunku, od przodu lub od tyłu.

Metoda nie wymaga żadnej fazy przed rozpoczęciem wykonywania algorytmu. Ze względu na charakter metody wymaga dodatkowego miejsca w pamięci. Złożoność czasowa wynosi $O(m*n)$, a ilość operacji potrzebna do wykonania algorytmu wynosi $2n$.

Przykładowy kod źródłowy algorytmu brute – force, wyszukującego określony wzorzec w podanym tekście:

```
void BF(char *x, int m, char *y, int n) {
    int i, j;
    for (j = 0; j <= n - m; ++j) {
        for (i = 0; i < m && x[i] == y[i + j]; ++i);
```

```
        if (i >= m)
            OUTPUT(j);
    }
}
```

4.2 Deterministyczny automat skończony

DFA (Deterministic Finite Automaton), czyli deterministyczny automat skończony to maszyna o skończonej liczbie stanów, która czytając kolejne symbole podanego słowa zmienia swój stan na wartość funkcji opisującej dany symbol. Po przeczytaniu całego słowa automat sprawdza, czy znajduję się w jednym z określonych stanów akceptacyjnych.

Algorytm zbudowany na podstawie DFA na początku wymaga stworzenia minimalnego skończonego automaty $A(x)$ rozpoznającego język $\sum^* x$, w celu znalezienia słowa x. Minimalny automat określony wzorem $A(x) = (Q, q_0, T, E)$ rozpoznający język $\sum^* x$ definiujemy następująco:

- Q to zbiór wszystkich przedrostków słowa x:
  $Q = \{\epsilon, x[0], x[0..1], .., x[0..m-2]\}$, gdzie *epsilon* – zbiór pusty
- $q_0 = \epsilon$
- $T = \{x\}$
- dla q należącego do Q (q to przedrostek x) i a w $\sum$ (q, a, qa) należy do $\sum$ wtedy i tylko wtedy, gdy qa jest także przedrostkiem słowa x, w przeciwnym razie (q, a, p) należy do $\sum$, a p jest najdłuższym przyrostkiem qa, który jest przedrostkiem słowa x

Po stworzeniu DFA przeszukiwanie słowa x w tekście y polega na przeanalizowaniu tekstu y przez DFA rozpoczynając ze stanem q_0. Za każdym razem, gdy trafiono na terminal (T - szukane słowo) raportowane jest wystąpienie szukanego słowa.

Stworzenie DFA $A(x)$ ma złożoność czasową rzędu $O(m+\sigma)$ i potrzebuje $O(m\sigma)$ miejsca, natomiast sam etap przeszukiwania ma złożoność czasową $O(n)$.

Przykładowa implementacja DFA:

```
void preAut(char *x, int m, Graph aut) {
   int i, state, target, oldTarget;
   for (state = getInitial(aut), i = 0; i < m; ++i) {
      oldTarget = getTarget(aut, state, x[i]);
      target = newVertex(aut);
      setTarget(aut, state, x[i], target);
      copyVertex(aut, target, oldTarget);
      state = target;
   }
   setTerminal(aut, state);
}

void AUT(char *x, int m, char *y, int n) {
   int j, state;
   Graph aut;
   /* Preprocessing */
   aut = newAutomaton(m + 1, (m + 1)*ASIZE);
   preAut(x, m, aut);
   /* Searching */
   for (state = getInitial(aut), j = 0; j < n; ++j) {
      state = getTarget(aut, state, y[j]);
      if (isTerminal(aut, state))
         OUTPUT(j - m + 1);
   }
}
```

4.3 Algorytm Karp-Rabin

Algorytm Karpa-Rabina jest kolejnym algorytmem służącym do przeszukiwania określonego podciągu w tekście. Algorytm ten korzysta z funkcji hashującej, co jest prostą metodą, aby w większości przypadków uniknąć kwadratowej liczby porównań znakowych. Zamiast sprawdzania znaku na każdej pozycji tekstu, w celu odnalezienia wzorca, bardziej wydajne jest sprawdzenie, czy zawartość "okna" przypomina wzorzec. Sprawdzanie podobieństwa pomiędzy wzorcem i "oknem" odbywa się za pomocą funkcji hashującej.

Właściwości funkcji hashującej:

- Wydajność obliczeń
- Dyskryminuje słowa
- funkcja $hash(y[j+1..j+m])$ musi być łatwo przeliczalna z $hash(y[j..j+m-1])$ oraz $y[j+m]$:
  $hash(y[j+1..j+m]) = rehash(y[j], y[j+m], hash(y[j..j+m-1]))$

Dla słowa w o długości m niech funkcja $hash$ będzie zdefiniowana następująco:

$$hash(w[0..m-1]) = (w[0] * 2^{m-2} + ... + w[m-1] * 2^0) mod(q),$$
gdzie q jest dowolną dużą liczbą

Wtedy funkcja rehash przyjmuje postać:

$$rehash(a, b, h) = ((h - a * 2^{m-2}) * 2 + b) mod(q)$$

Preprocessing - obliczenie funkcji hashującej dla szukanego wzorca - posiada złożoność czasową rzędu $O(m), gdziem - dugowzorca.$

Podczas wyszukiwania wystarczy porównać $hash(x)$ z $hash(y[j..j+m-1])$ dla $0 <= j <= n-m$. Jeśli równość jest znaleziona to należy sprawdzić równość $x = y[j..j+m-1]$ znak po znaku.

Przykład implementacji:

```
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))

void KR(char *x, int m, char *y, int n) {
    int d, hx, hy, i, j;

    /* Preprocessing */
    /* computes d = 2^(m-1) with
       the left-shift operator */
    for (d = i = 1; i < m; ++i)
       d = (d<<1);

    for (hy = hx = i = 0; i < m; ++i) {
       hx = ((hx<<1) + x[i]);
       hy = ((hy<<1) + y[i]);
    }

    /* Searching */
    j = 0;
    while (j <= n-m) {
       if (hx == hy && memcmp(x, y + j, m) == 0)
          OUTPUT(j);
       hy = REHASH(y[j], y[j + m], hy);
       ++j;
    }

}
```

4.4 Algorytm Boyer-Moore

Algorytm Boyer-Moore został stworzony w 1977 roku i do tej pory uważany jest za najbardziej optymalny algorytm poszukiwania wzorca w tekscie. Dowodem na to jest fakt, że algorytm ten jest zaimplementowany w większości najbardziej popularnych aplikacji z opcją „Znajdź" lub „Zamień".

Algorytm szuka danego wzorca w teksćie porównując kolejne znaki tekstu do wzorca, lecz w odróżnieniu od metody brute-force wykorzystuje informacje zebrane za pomocą fukcji preprocesujących, aby pominąć jak najwięcej znaków w tekście, które na pewno nie pasują do wzorca.

Procedura zaczyna się w miejscu k = n, w taki sposób, że początek wzorca P jest wyrównany do początku tekstu T. Następnie poszczególne znaki z P i T są porównywane począwszy od indeksu n w P i indeksu k w T, poruszając się od prawej do lewej strony wzorca P. Porównywanie trwa dopóki znaki w P i T nie są różne lub do momentu dotarcia do początku P (co oznacza, że znaleziono wystąpienie w tekście), a następnie indeks porównywania przesuwany jest w prawo o maksymalną wartość wskazywaną przez zdefiniowane reguły. Czynności są powtarzane do momentu sprawdzenia całego tekstu T.

Reguły przesuwania indeksu definiowane są za pomocą tabel tworzonych za pomocą funkcji preprocesujących.

Przykład implementacji:

```
public List<Integer> match() {
        List<Integer> matches = new LinkedList<Integer>();
        computeLast();
        computeMatch();

        int i = text.length() - 1;
        int j = pattern.length() - 1;
        while (i >= 0 && i < text.length()) {
                try {
                        if (pattern.charAt(j) == text.charAt(i)) {
```

```
                        if (j == 0) {
                                matches.add(i);
                                j = pattern.length() - 1;
                                // return i;
                        }
                        j--; i--;
                    } else {
                        i -= Math.max(match[j], last[text.charA
                        j = pattern.length() - 1;
                    }
                } catch (Exception ex) { (...) }
        }
        return matches;
}
```

5 Wprowadzenie do Asemblera

6 Projekt systemu CodeComp

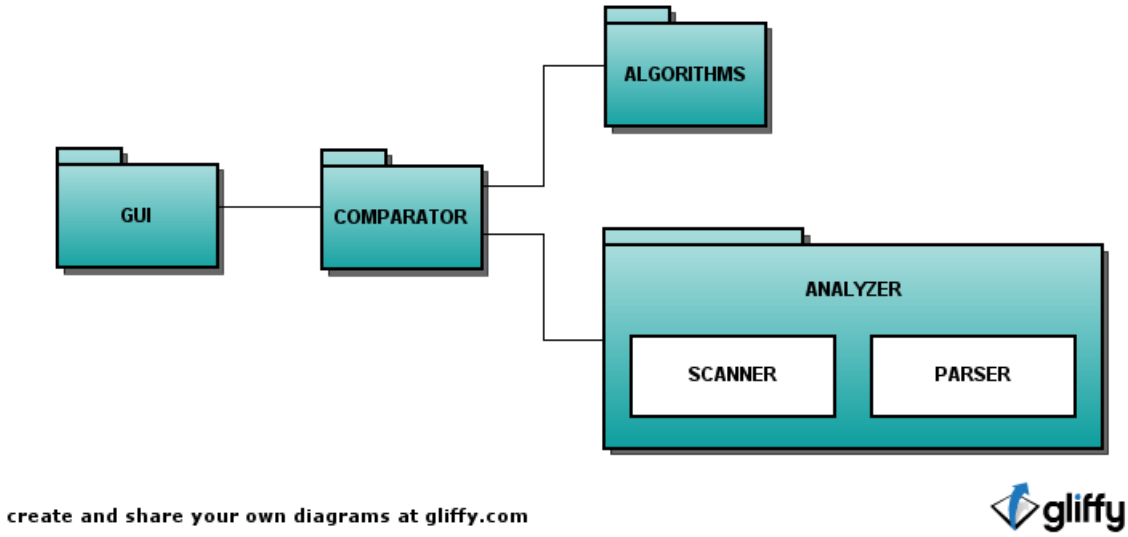
W tym rozdziale przedstawiono opis aplikacji CodeComp, jej budowę wewnętrzną oraz sposoby instalacji i użycia.

6.1 Architektura systemu

Program składa się z czterech głownych pakietów: Comparator, Algorithm, Analyzer oraz GUI.

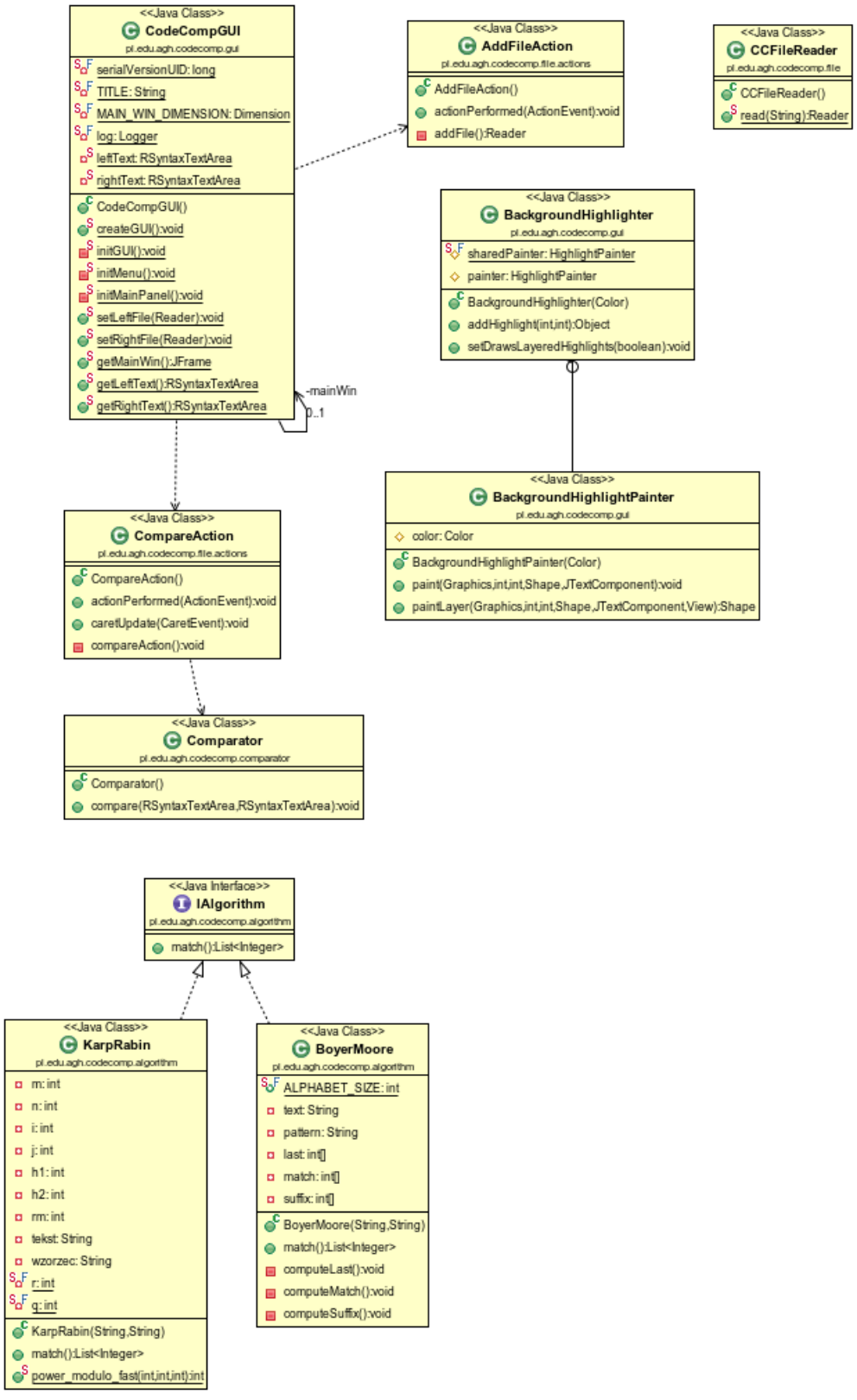
Comparator jest podstawowym, statycznym obiektem, który na wejście pobiera kody źrodłowe programów wejściowych, uruchamia wybrany sposób porównywania i zwraca listę elementów podobnych oraz ich położenia w tekście.

Algorithm to pakiet, w którego skład wchodzi zestaw algorytmów porównywania tekstu. Użytkownik jest w stanie wybrać sobie wybrany algorytm w zależności od

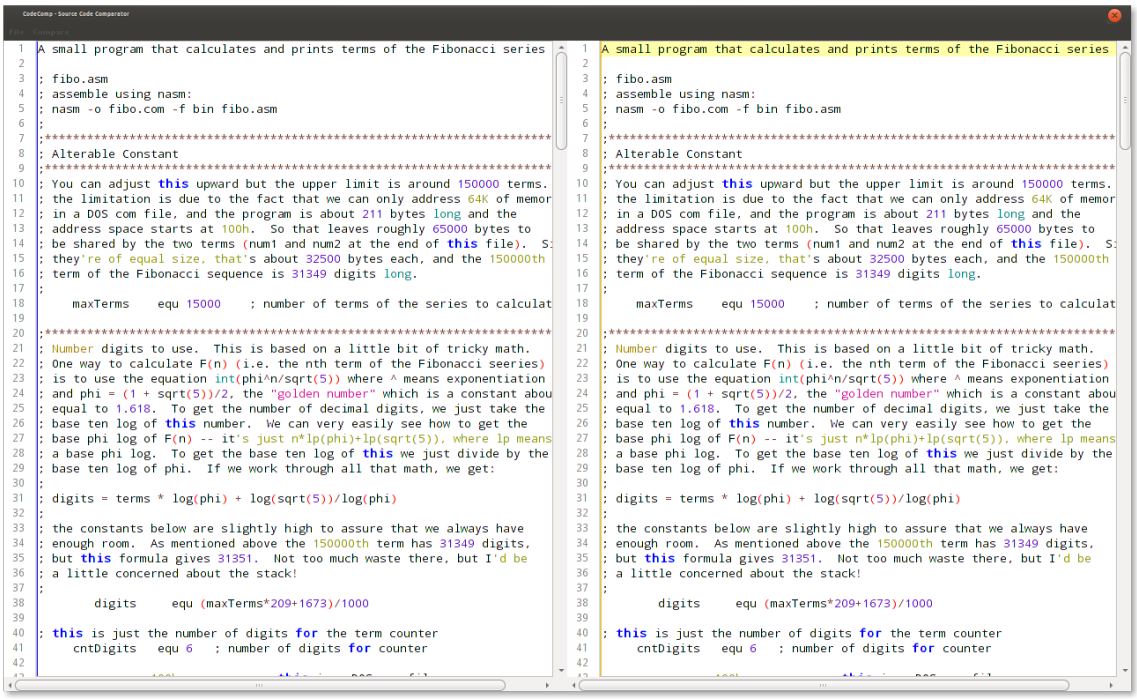


potrzeb, aby móc porównać jego skuteczność oraz czas działania.

Analyzer to komponent, który zajmuje się etapem analizy z teorii kompilacji. Składa się ze skanera - czyli analizatora leksykalnego - oraz z parsera - czyli analizatora składniowego. Wynikiem działania analizatora jest drzewo składniowe, które następnie będzie porównywane przez komparator.


<<Java Class>>
CodeCompGUI
pl.edu.agh.codecomp.gui
serialVersionUID: long
TITLE: String
MAIN_WIN_DIMENSION: Dimension
log: Logger
leftText: RSyntaxTextArea
rightText: RSyntaxTextArea
CodeCompGUI()
createGUI():void
initGUI():void
initMenu():void
initMainPanel():void
setLeftFile(Reader):void
setRightFile(Reader):void
getMainWin():JFrame
getLeftText():RSyntaxTextArea
getRightText():RSyntaxTextArea
-mainWin
0..1
<<Java Class>>
AddFileAction
pl.edu.agh.codecomp.file.actions
AddFileAction()
actionPerformed(ActionEvent):void
addFile():Reader
<<Java Class>>
CCFileReader
pl.edu.agh.codecomp.file
CCFileReader()
read(String):Reader
<<Java Class>>
BackgroundHighlighter
pl.edu.agh.codecomp.gui
sharedPainter: HighlightPainter
painter: HighlightPainter
BackgroundHighlighter(Color)
addHighlight(int,int):Object
setDrawsLayeredHighlights(boolean):void
<<Java Class>>
BackgroundHighlightPainter
pl.edu.agh.codecomp.gui
color: Color
BackgroundHighlightPainter(Color)
paint(Graphics,int,int,Shape,JTextComponent):void
paintLayer(Graphics,int,int,Shape,JTextComponent,View):Shape
<<Java Class>>
CompareAction
pl.edu.agh.codecomp.file.actions
CompareAction()
actionPerformed(ActionEvent):void
caretUpdate(CaretEvent):void
compareAction():void
<<Java Class>>
Comparator
pl.edu.agh.codecomp.comparator
Comparator()
compare(RSyntaxTextArea,RSyntaxTextArea):void
<<Java Interface>>
IAlgorithm
pl.edu.agh.codecomp.algorithm
match():List<Integer>
<<Java Class>>
KarpRabin
pl.edu.agh.codecomp.algorithm
m: int
n: int
i: int
j: int
h1: int
h2: int
rm: int
tekst: String
wzorzec: String
r: int
q: int
KarpRabin(String,String)
match():List<Integer>
power_modulo_fast(int,int,int):int
<<Java Class>>
BoyerMoore
pl.edu.agh.codecomp.algorithm
ALPHABET_SIZE: int
text: String
pattern: String
last: int[]
match: int[]
suffix: int[]
BoyerMoore(String,String)
match():List<Integer>
computeLast():void
computeMatch():void
computeSuffix():void

Program posiada graficzny interfejs użytkownika stworzony za pomocą bilbioteki Swing. Składa się on z dwóch elementów tekstowych wyświetlających wczytane kody źródłowe oraz podświetlający elementy, które są identyczne/podobne. Wczytywanie oraz wszelkie inne operacje dostępne są z poziomu głównego menu znajdującego się na górze okna.



6.2 Uruchamianie aplikacji

Aplikacja dołączona do niniejszej pracy dyplomowej w całości napisana została w języku JAVA. Wykorzystuje ona podstawowe pakiety ze środowiska JRE 1.7 oraz kilka dodatkowych, niestandardowych bibliotek (rsyntaxtextarea).

Aplikacja dołączona została w postaci kodu źrodłowego oraz skompilowanego archiwum JAR, który można uruchomić na dowolnym środowisku z zainstalowanym pakietem uruchomieniowym JRE w wersji 1.7 za pomocą polecenia:

java -jar codecomp.jar

6.3 Korzystanie z aplikacji

6.4 Wyniki

7 Wnioski

8 Bibliografia

Literatura

[1] ldots