TKOM - Dokumentacja projektowa

Krzysztof Najda

Maj 2019

1 Funkcjonalność

Program, który analizuje kod w podzbiorze C++ i znajduje nieużywane elementy programu: zmienne, stałe, klasy, funkcje, metody, konstruktory, etc. .
Można zdefiniować ograniczenia do rozpatrywanego kodu.

2 Dopuszczalne konstrukcje

Dopuszczalne typy: int, double, bool, void.
Możliwe definiowanie klas, ale bez dziedziczenia i klas zagnieżdzonych.
Klasy mogą mieć częśc publiczną (public) i prywatną (private).
Brak tablic, pointerów i referencji, co implikuje brak konstruktorów kopiujących.
Możliwe bloki warunkowe (if else) oraz pętle (while) z zagnieżdzeniami.
Funkcje mogą być przeciążane.

if	else	while	return
void	bool	int	double
class	private	public	
true	false		

Tabela 1: Słowa kluczowe

2.1 Operatory arytmetyczne

```
a = -b;    //wartosc przeciwna
a = b * c; //mnozenie
a = b \ c; //dzielenie
a = b + c; //dodawanie
a = b - c; //odejmowanie
a = b % c; //modulo
```

2.2 Operatory relacyjne

```
a == b //rowne
a != b //rozne
a >= b //wieksze rowne
a <= b //mniejsze rowne
a < b  //mniejsze
a > b  //wieksze
```

2.3 Operatory boolowskie

```
!a     //negacja
a || b //or
a && b //and
```

2.4 Zmienne

```
int a12, bfd, cv1;
double f_48;
bool _yug, bbb;

a12 = 58;
f_48 = -8.15;
_yug = false;
```

2.5 Funkcja

```
void func(int arg1, int arg2 /* list of arguments */)
{
    /* statements */
}
```

2.6 Blok warunkowy

```
if(/*expression*/)
{
    /* statements */
}
else
{
    /* statements */
}

if(/*expression*/) /* statement */;
else /* statement */;
```

2.7 Pętla

```
while(/*expression*/)
{
    /* statements */
}

while(/*expression*/) /* statement */;
```

2.8 Klasa

```
class exampleClass
{
    private:
        int a, b;
```

```
        double foo(double c)
        {
            a = a * c;
            return a + b * c;
        }

    public:
        double d;

        double boo()
            return (a + b) * foo(d);

        exampleClass(int a_, int b_)
        {
            a = a_;
            b = b_;
            d = a * b;
        }
};
```

3 Specyfikacja i składnia

```
digit      =   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

letter     = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l"
    | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y"
    | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L"
    | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y"
    | "Z"

ident      =   letter | "_" {letter | "_" | digit}

type       =   ident

const      =   constint | constdouble

constint   =   digit {digit}

constdouble =  digit {digit} "." digit {digit}

constbool  =   "true" | "false"

comment    =   simple_com | comp_com

simple_com =   "//" anything-"\n" "\n"

comp_com   =   "/*" anything-"*/" "*/"

rel_op     =   "==" | "<" | ">" | "<=" | "!=" | ">="

sign       =   "+" | "-"
```

```
mul_op     =   "*" | "/" | "%"

add_op     =   "+" | "-"

program    =   {s_variable | s_function | s_class}

s_variable =   type ident {"," ident} ";"

s_function =   type ident "(" [type ident {"," type ident}] ")" block_st

s_class    =   "class" ident "{" class_content "}" ";"

class_content =   {private_part | public_part}

private_part  =   "private:" {s_variable | s_function | s_constructor}

public_part   =   "public:" {s_variable | s_function | s_constructor}

s_constructor =   type "(" [type ident {"," type ident}] ")" block_st

c_ident    =   ident {"." ident}

function   =   c_ident "(" [expr {"," expr}] ")"

statement  =   simple_st | block_st

simple_st  =   s_variable | if_st | while_st | s_simple_expr | assignment |
    return_st

block_st   =   "{" {s_variable | if_st | while_st | s_simple_expr | assignment |
    return_st} "}"

if_st      =   "if" "(" bool_expr ")" statement ["else" statement]

while_st   =   "while" "(" bool_expr ")" statement

expr       =   simple_expr | comp_expr | bool_expr

s_simple_expr = simple_expr ";"

comp_expr  =   simple_expr rel_op simple_expr

simple_expr =  simple_expr2 {add_op simple_expr2}

simple_expr2 = arithm_ele {mul_op arithm_ele}

arithm_ele =   [sign] (c_ident | function | const)

bool_expr  =   bool_expr2 {"||" bool_expr2}

bool_expr2 =   bool_ele {"&&" bool_ele}

bool_ele   =  ["!"] (c_ident | function | constbool | comp_expr)
```

```
assignment =   c_ident "=" expr ";"

return_st  =   "return" [expr] ";"
```

4 Sposób uruchomienia

```
./tkom [-p][-s] nazwa_pliku
```

Wybrany przez użytkownika plik wejściowy zostanie przeanalizowany przez program.

Opcja -s wyświetli dodatkowe informacje dotyczące analizy semantycznej: zadeklarowane zmienne, funkcje, klasy oraz ile razy zostały użyte. Opcja -p wyświetli drzewo wygenerowane przez parser.

Jeśli podany kod nie jest poprawny, program wypisze odpowiedni komunikat, zawierający takie informacje jak rodzaj błędu i numer linijki w której ten błąd wystąpił.

Jeśli natomiast analiza kodu przebiegnie pomyślnie program wypisze po kolei wszystkie nieużywane zmienne, stałe, klasy, konstruktory itp. wraz z numerami linijek w których się znajdują.

5 Przykłady

5.1 Nieużywane stałe

Stałe zostają uznane za nieużywane jeśli występują jako samotne wyrażenie tj. nie są częścią przypisania, zwracania, argumentem funkcji albo wyrażeniem warunkowym w if lub while.

```
1 void func()
2 {
3     int a;
4     a=5;
5     5;  //Line 5: Unused constant
6 }
```

Listing 1: Nieużywana stała

```
1 void func()
2 {
3     int a;
4     a=5;
5     a+5*a; //Line 5: Unused expression
6 }
```

Listing 2: Nieużywane wyrażenie

5.2 Nieużywane zmienne

Zmienna jest uznawana za nieużywaną jeśli została zadeklarowana, ale potem nie zostały wykonane żadne akcje związane z nią. Jeśli zmienna zostanie zainicjalizowana, albo użyta argument funkcji/konstruktora, nie pojawi się ostrzeżenie o nieużywanej zmiennej. W przypadku instancji klas zadziała również odwołanie do którejkolwiek zmiennej lub funkcji należącej do tej klasy.

```
int func()
{
    int arg3; //line 3: Unused
        variable 'arg3'
    exampleClass ex; //line 4: Unused
        variable 'ex'
    return 5;
}
```

Listing 3: Nieużywane zmienne

```
int func()
{
    int arg3;
    exampleClass ex;
    arg3 = ex.func();
    return arg3;
}
```

Listing 4: Brak ostrzeżenia

Również argument funkcji może zostać uznany za nieużywany, jeśli nie występuje w ciele funkcji.

```
int func(int arg3) //line 1: Unused
    variable 'arg3'
{
    return 5;
}
```

Listing 5: Nieużywany argument

```
int func(int arg3)
{
    return arg3;
}
```

Listing 6: Brak ostrzeżenia

Jeśli zmienna należy do klasy, a dostęp do niej jest prywatny, to przynajmniej jedna z metod wewnątrz tej klasy musi użyć tej zmiennej.

Jeśli dostęp do zmiennej jest publiczny, to musi zostać wykorzystana w jakimkolwiek miejscu programu, aby zostać uznaną za używaną.

```
class example
{
    private:
        int value; //line 4: Unused
            variable 'value'
    public:
        int func(int arg)
        {
            return arg * 2;
        }
};
```

Listing 7: Nieużywana zmienna wewnątrz klasy

```
class example
{
    private:
        int value;
    public:
        int func(int arg)
        {
            return arg * value;
        }
};
```

Listing 8: Brak ostrzeżenia

5.3 Nieużywane funkcje

Funkcja jest uznawana za nieużywaną jeśli została ona zdefiniowana, ale nigdzie nie wystąpiło jej wywołanie.

```
int foo(int arg) //Line 1: Unused
    function 'foo'
{
    return 3 * arg;
}

int main()
{
    int a;
    a = 3;
    return 0;
}
```

Listing 9: Nieużywana funkcja

```
int foo(int arg)
{
    return 3 * arg;
}

int main()
{
    int a;
    a = foo(2);
    return 0;
}
```

Listing 10: Brak ostrzeżenia

Funkcje mogą być przeciążane, wtedy numer linijki pozwala jednoznacznie zidentyfikować funkcję która spowodowała pojawienie się ostrzeżenia.

```
int foo(int arg)
{
    return 2 - arg;
}

int foo(bool arg) //Line 8: Unused function 'foo'
{
    if(arg) return 5;
    else return -5;
}

int main()
{
    int a;
    a = foo(15);

    return 0;
}
```

Listing 11: Przeciążone funkcje

Funkcje, tak samo jak inne konstrukcje, są uznawane za używane nawet jeśli ich jedyne wywołanie nastąpiło w kontekście który został uznany za nieużywany. Analiza wykorzystania danej konstrukcji, nie idzie zatem w głąb, co dobrze widać na poniższym przykładzie.

```
int a;

int foo(int arg)
{
    return a * arg;
}

void bar() //Line 8: Unused function 'boo'
{
    a = foo(10);
}

int main()
{
    a = 7;
    return 0;
}
```

Zgodnie ze wcześniejszymi założeniami funkcja bar(), jest w tym przypadku uznana za nieużywaną. Natomiast funkcja foo() została uznana za używaną, chociaż została wywołana tylko i wyłącznie w nieużywanej funkcji bar().

5.4 Nieużywane klasy

Klasa jest uznawana za nieużywaną jeśli poza ciałem klasy nie została zadeklarowana jej instancja.

```
class example //Line 1: Unused class
    'example'
{
    private:
        double val;
    public:
        double getVal() //Line 6:
            Unused function 'getVal'
        {
            return val;
        }
};

int main(int arg)
{
    return 0;
}
```

Listing 12: Nieużywana klasa

```
class example
{
    private:
        double val;
    public:
        double getVal() //Line 6:
            Unused function 'getVal'
        {
            return val;
        }
};

int main(int arg)
{
    example ex; //Line 15: Unused
        variable 'ex'
    return 0;
}
```

Listing 13: Nieużywana zmienna

Jak widać, mimo deklaracji instancji klasy, nie pozbyliśmy się ostrzeżeń. Stało się tak ponieważ zmienna ex nie została explicite zainicjalizowana, co spowodowało pojawienie się ostrzeżenia o nieużywanej zmiennej. Użycie konstruktora wprost sprawiłoby że ostrzeżenie to nie pojawiłoby się.

Oprócz tego w obu przypadkach występuje ostrzeżenie o nieużywanej funkcji getVal. Zarówno ostrzeżenie o nieużywanej zmiennej, jak i o nieużywanej funkcji nie zostałyby pokazane, gdyby wywołano metodę ex.getVal().

```
1  class example
2  {
3      private:
4          double val;
5      public:
6          double getVal() //Line 6:
               Unused function 'getVal'
7          {
8              return val;
9          }
10 };
11
12 int main(int arg)
13 {
14     example ex;
15     ex = example();
16     return 0;
17 }
```

Listing 14: Użycie konstruktora wprost

```
1  class example
2  {
3      private:
4          double val;
5      public:
6          double getVal()
7          {
8              return val;
9          }
10 };
11
12 int main(int arg)
13 {
14     example ex;
15     ex.getVal();
16     return 0;
17 }
```

Listing 15: Brak ostrzeżeń

5.5 Nieużywane konstruktory

Konstruktory klas są uznawane za nieużywane, podobnie jak funkcje, jeśli nie zostały wywołane. Jedynym wyjątkiem jest konstruktor domyślny, który może być zawołany nie wprost podczas deklaracji zmiennej.

Również podobnie do funkcji konstruktor może być zawołany tylko w nieużywanej konstrukcji, ale nie spowoduje to uznania danego konstruktora za nieużywany.

```
class exampleClass //Line 1: Unused class 'exampleClass'
{
    private:
        bool val;

    public:
        exampleClass()
        {
            val = false;
        }
        exampleClass(bool b)
        {
            val = b;
        }
        void foo()
        {
            exampleClass ex;
            ex = exampleClass(false);
        }
};
int main()
{
    return 0;
}
```

Listing 16: Brak ostrzeżeń o nieużywanym konstruktorze

```
class exampleClass
{
    private:
        int val;

    public:
        exampleClass()
        {
            val = 5;
        }

        exampleClass(int a)   //line 12: Warning unused constructor 'exampleClass'
        {
            val = a;
        }
};

int main()
{
    exampleClass a;   //line 20: Warning unused variable 'a'
}
```

Listing 17: Nieużywany konstruktor

```
class exampleClass
{
    private:
        int val;

    public:
        exampleClass()
        {
            val = 5;
        }

        exampleClass(int a)
        {
            val = a;
        }
};

int main()
{
    exampleClass a;
    a = exampleClass(15);
}
```

Listing 18: Brak ostrzeżeń

6 Realizacja

Program będzie się składać z kilku modułów:

- Źródło (obsługa plików wejściowych)
- Analizator leksykalny
- Analizator składniowy
- Analizator semantyczny
- Generator ostrzeżeń o nieużywanych konstrukcjach
- Menadżer obsługi błędów i ostrzeżeń

Będą także istnieć inne struktury danych:

- Tokeny
- Drzewo wyprowadzenia
- Tablica symboli
- Lista typów
- Komunikaty o błędach/ostrzeżeniach

6.1 Źródło

Klasa znajdująca się na samym początku łańcucha modułów, dająca abstrakcje pliku wejściowego.

W szeczgólności umożliwia sekwencyjny dostęp do kolejnych symboli ze źródła, za pomocą odpowiedniej metody. Moduł ten śledzi także numer obecnej linijki i wystawia metodę która umożliwia odczytanie tej wartości.

6.2 Analizator leksykalny

Kolejnym modułem jest analizator leksykalny, którego zadaniem jest tworzenie tokenów, które następnie są konsumowane przez parser.

Token składa się z:

- Rodzaju tokenu;
- Pewnej wartości (np. dla tokenu reprezentującego identyfikator - string z nazwą identyfikatora, a dla tokenu reprezentującego wartość całkowitoliczbową - int);
- Numeru linijki w której dany token się zaczyna;

Lekser tworzy nowy token pobierając ze źródła kolejne symbole oraz numer linijki pierwszego symbolu należącego do tokenu.

6.3 Parser

Analizator składniowy pobiera z leksera sekwencyjnie kolejne tokeny, z których tworzy drzewo wyprowadzenia. Parser jest typu LL(n). W węzłach drzewa znajdować się będą odpowiednie reguły gramatyczne. Symbolem startowym takiego drzewa jest zawsze reguła 'Program'.

W trakcie próby utworzenia drzewa wyprowadzenia, sprawdza on czy strumień tokenów jest zgodny z założoną gramatyką - jeśli nie jest to wysyła on odpowiedni komunikat do menadżera błędów.

6.4 Analizator semantyczny

Ten moduł ma na wejściu drzewo wyprowadzenia wygenerowane przez parser i sprawdza czy drzewo to jest poprawne semantycznie:

- Sprawdza poprawność typów
- Sprawdza czy dana zmienna/funkcja nie została zadeklarowana więcej niż jeden raz
- Sprawdza czy dana zmienna/funkcja została zadeklarowana
- Utrzymuje tablice symboli i listę typów

Wszystkie funkcje/zmienne/klasy przetrzymywane są w kontenerze asocjacyjnym typu std::map. Kluczem jest nazwa (w przypadku zmiennej lub klasy), lub prototyp (w przypadku funkcji).

Oprócz tego utrzymywana jest lista typów, która zawiera wszystkie domyślne typy (int, double, bool), jak i nazwy klas obecnych w programie.

Również w tym module odbywa się analiza nieużywanych konstrukcji zgodnie z zasadami opisanymi w punkcie 5.

6.5 Menadżer błędów

Każdy z analizatorów jest podłączony do menadżera obsługi błędów i może wysyłać do niego dwa rodzaje komunikatów: błąd lub ostrzeżenie. Różne moduły wysyłają różne rodzaje błędów/ostrzeżeń:

- Analizator leksykalny - niezgodne z przepisami na tokeny ciągi znaków;
- Analizator składniowy - niezgodne z założoną gramatyką ciągi tokenów;
- Analizator semantyczny - niezgodne z założonymi regułami konstrukcje gramatyczne;
- Generator ostrzeżeń - nieużywane konstrukcje - zmienne, funkcje, klasy i konstruktory;

W przypadku wystąpienia błędu, dalsze działanie programu jest wstrzymywane, a odpowiedni komunikat o błędzie wraz z numerem linii wktórej wystąpił jest wysyłany na standardowe wyjście. Jeśli do menadżera zostało wysłane ostrzeżenie, a nie błąd to przechowuje on je i pozwala programowi się dalej wykonywać - komunikat o ostrzeżeniu zostanie wyświetlony po zakończeniu działania programu.

Moduły wysyłają do menadżera jedynie informacje o typie błędu (enum) i numerze linijki w której wystąpił - to zadaniem menadżera jest zinterpretowanie tej informacji i wyświetlenie na standardowym wyjściu odpowiedniego komunikatu dla użytkownika.

7 Testowanie

Zostaną przygotowane różne pliki testowe, niektóre zawierające różnego rodzaju błędy, które powinny być wykryte przez program. Jeśli dany plik przeszedł bez błędów analizę programem, to taki plik powinien być również kompilowalny jednym ze znanych kompilatorów c++, np. gcc. Jeśli zatem program nie generuje błędów dla jakiegoś pliku, a mimo to plik ten nie daje się skompilować za pomoca gcc, to oznacza to że program nie jest kompletny lub zawiera jakiś błąd.

W podobny sposób zostanie przetestowana główna funkcjonalność programu, jaką jest wykrywanie nieużywanych konstrukcji. Pewne pliki testowe, kompilowalne za pomocą gcc, zostaną podane programowi do analizy. Po otrzymaniu z programu informacji o nieużywanych konstrukcjach zostaną one ręcznie usunięte z plików testowych. Jeśli takie pliki również powinny kompilować się za pomocą gcc, a rezultat ich wykonania być taki sam jak przed usunięciem konstrukcji uznanych za nieużywane.