Treść zadania:

Zad O2

Napisać program, który optymalizuje kod w C (podzbiór C, bez pointerów etc…). Optymalizacja polega na usunięciu niekoniecznych obliczeń z wnętrz pętli. Zdefiniować składnię kodu wyrażeń które podlegają optymalizacji. Można przyjąć ograniczenia dotyczące użycia w kodzie wskazań na funkcje itp. Możliwe zagnieżdżenie pętli

Opis zakładanej funkcjonalności:

- Odczyt, parsowanie i analiza kodu z plików tekstowych w języku C
- Sprawdzanie poprawności kodu i zgłaszanie błędów
- Możliwość użycia oraz optymalizacja pętli for przez usunięcie z pętli niekoniecznych obliczeń
- Możliwość użycia instrukcji warunkowych i wyrażeń logicznych ||, &&, ==, !=, <, >, <=, >=, !
- Możliwość użycia operatorów matematycznych +, -, /, *
- Możliwość użycia typów int, float, double, long, short, bool

Przykłady:

1. Wyciągnięcie części mnożenia z pętli:

```
int main()
{
	int i, a = 5, b[100], c[100];
	for (i=1; i<100; i=i + 1)
	{
		b[i] = c[i] *a * 135.8;
	}
	return 0;
}
```

Kod po przekształceniu:

```
int main()
{
	int i, a = 5, b[100], c[100];
	float gen1 = a*135.8;
	for (i=1; i<100; i= i + 1)
	{
		b[i] = c[i] * gen1;
	}
	 return 0;
 }
```

2. Wyciągnięcie kilku obliczeń z pętli:

```
int main()
{
   int a =5;
   int b = 3;
   int c = 9;
   for (int i=1; i<10; i = i + 1)
   {
      b = 2 * a - 7/9;
      c = 7+(7)/b;
   }
   return 0;
}
```

Kod po przekształceniu:

```
int main()
{
          int a = 5;
          int b = 3;
          int c = 9;
          int gen0 = 2 * a - 7 / 9;
          int gen1 = (7);
          for(int i = 1;i < 10; i = i + 1)
          {
                    b =  + gen0;
                    c = 7 + 1 / b * gen1;
          }
          return 0;
}
```

3. Wyciągnięcie mnożenia z zagnieżdżonej pętli:

```
int main()
{
          int i, j, a=5, b[100], c[100];
          for (i=1; i<100; i= i + 1)
          {
              for (j=1; j<100; j = j + 1)
               {
                    b[i] = c[j] * (a *4 + b[20]);
               }
          }
          return 0;
}
```

Kod po przekształceniu:

```
int main()
{
        int i,  j,  a = 5,  b[100],  c[100];
         int gen0 = a * 4;
        for(i = 1;i < 100; i = i + 1)
         {
                  for(j = 1;j < 100; j = j + 1)
                 {
                          b[i] = c[j] * (gen0 + b[20]);
                  }
         }
         return 0;
}
```

4. Wyciągnięcie mnożenia z zagnieżdżonej pętli I głównej pętli:

```
int main()
{
         int i, j, a=5, b[100], c[100];
        for (i=1; i<100; i= i + 1)
        {
                  for (j=1; j<100; j = j + 1)
                 {
                          a = a + 1;
                          b[j] = b[j] + c[i] * a *4 + 7*21;
                 }
        }
        return 0;
}
```

Kod po przekształceniu:

```
int main()
{
         int i,  j,  a = 5,  b[100],  c[100];
         int gen1 = 7 * 21;
         for(i = 1;i < 100; i = i + 1)
         {
                  int gen0 = c[i] * 4;
                  for(j = 1;j < 100; j = j + 1)
                 {
                           a = a + 1;
                           b[j] = b[j] + a * gen0 + gen1;
                  }
        }
        return 0;
}
```

5. Brak możliwości wyciągnięcia operacji:

```
int main()
{
        int i, j;
        float a=7.51;
        for (i=1; i<10; i++)
        {
                a = a + 1;
                for (j=1; j<10;j++)
                {
                        a = a * 2;
                        if(a > 1000000) break;
                        b[i] = c[j] * a *4;
                }
        }
        return 0;
}
```

Brak przekształceń w kodzie – zmienne wykorzystywane w operacji ulegają zmianie w każdej iteracji pętli;

6. Wyciągnięcie części operacji logicznej

```
int main()
{
   int i, j, a=5, b[100], c[100];
   for (i=1; i<100; i= i + 1)
   {
      for (j=1; j<100; j = j + 1)
      {
      int f = 4;
         if(a + 1> 25*6/f)
         {
            continue;
         }
      }
   }
   return 0;
}
```

Kod po przekształceniu:

```
int main()
{
	int i,  j,  a = 5,  b[100],  c[100];
	int gen0 = a + 1;
	int gen1 = 25 * 6;
	for(i = 1;i < 100; i = i + 1)
	{
		for(j = 1;j < 100; j = j + 1)
		{
			int f = 4;
			if( + gen0 > 1 / f * gen1)
			{
				continue;
			}
		}
	}
	return 0;
}
```

Gramatyka:

Program = int main() MainBlock

MainBlock = ”{“ { IfStatement | InitStatement | AssignStatement | ForStatement} “return” “0” “}”

StatementBlock= ”{“ { IfStatement | InitStatement | AssignStatement | ForStatement | ForInstruction } “}” ;

ForStatement = “for” “(“ [InitStatement | AssignStatement]”; ”[Expression] ”; ”[AssignStatement] ”) ” ”{“ StatementBlock “}”;

IfStatement = “if” “(“ Expression “)” **(**StatementBlock | AssignStatement | ForInstruction)

[“else” **(**StatementBlock | AssignStatement | ForInstruction)];

InitStatement = Type {(Variable | ArrayVariable) [AssignOp (Expression | “{“ {Expression “,”} Expression “}” ”) “,”]}

(Variable | ArrayVariable) [AssignOp (Expression | “{“ {Expression} Expression “}”)] “;” ;

AssignStatement = Variable AssignOp Expression “;” ;

Expression = “(“ Expression “)” | Expression ExpOperator Expression | Value

Type = “int” | “float” | “double” | “long” | “short” | “bool”

Variable = Letter { Letter, Digit } ;

Index = ”[” Expression ”] ” ;

ArrayVariable = Variable Index ;

Value = ([“+”]|[“-“]) [NegationOp] (Variable | FiniteNumber | Number | ArrayVariable| LogicValue);

ExpOperator = RelationOp | LogicalOp | AdditiveOp | MultiplicativeOp;

RelationOp = ”> ” | ”< ” | ”>= ” | ”<= ” | ”== ” | ”!= ” ;

LogicalOp = ”|| ” | ”&& ” ;

NegationOp = “!” ;

AssignOp = “=” ;

AdditiveOp = ”+ ” | ”- ” ;

MultiplicativeOp = ”* ” | ”/ ” ;

LogicValue = “true” | “false”;

ForInstruction = (break | continue) ";"

Letter = "a"…"z" | "A"…"Z" ;

Digit = "0"… "9" ;

Number = {Digit};

FiniteNumber = Number "." {Digit}

Tokeny:

Main, For, Break, Continue, Return, EOF,
Int, Short, Long, Float, Double, Bool,
Plus, Minus, Multiply, Divide,
If, Else, Or, And, Negation,
Assign, Equal, NotEqual, Less, Greater, LessOrEqual, GreaterOrEqual,
OpenBrace, ClosedBrace, OpenCurlyBrace, ClosedCurlyBrace, OpenSquareBrace,
ClosedSquareBrace,
Identifier, Comma, Semicolon,
Number, FiniteNumber, True, False,
Unknown

Wymagania funkcjonalne:

- Odczytanie, parsowanie i analiza kodu zapisanego w pliku tekstowym
- Kontrola poprawności danych oraz zgłaszanie błędów
- Optymalizacja wykonania pętli przez usunięcie niekoniecznych obliczeń
- Generacja poprawnego kodu

Wymagania niefunkcjonalne:

- Program w sposób jasny wskazuje znalezione błędy,
- Zmiany w kodzie powstałe w skutek optymalizacji są w poprawny sposób ukazane w pliku wyjściowym.

Sposób uruchomienia:

Program dostaje na wejście z plik z kodem. Program komunikuje wyniki kolejnych etapów analizy z wskazaniem błędów, jeśli takie występują. Zoptymalizowany kod jest wypisywany do nowoutworzonego pliku tekstowego.

Sposób realizacji:

- Program pisany w języku Java
- Do testów jednostkowych wykorzystano Junit
- Moduły:
    - Moduł odczytu pliku – odczytuje znaki z pliku i przekazuje je do leksera
    - Moduł lexera – analiza leksykalna kodu, odpowiedzialna za podzielenie pliku wejściowego na tokeny. Odczyt będzie się odbywał znak po znaku do momentu do

momentu odczytania rozpoznawalnego tokenu języka. Zwracane będą one do parsera

- Moduł parsera – analiza składniowa kodu, pobierając tokeny z lexera sprawdza ich poprawność gramatyczną. Tworzy drzewo składniowe(niebinarne AST) w wypadku braku błędów. W przeciwnym wypadku wypisuje pierwszy napotkany błąd na standardowe wyjście
- Moduł analizatora semantycznego – sprawdza poprawność utworzonego przez parser drzewa. W swoim działaniu:
    - Sprawdza poprawność inicjalizacji zmiennych i ich początkowych wartości,
    - Sprawdza czy zmienne zostały użyte po zainicjalizowaniu,
    - Sprawdza brak użycia liczby niecałkowitej przy odwołaniu do tablicy,
    - Napotkane błędy wypisuje na standardowe wyjście
- Moduł optymalizacyjny – dokonuje zmian w drzewie, które przeszło przez poprzednie moduły w celu optymalizacji w pętlach for
- Moduł generujący kod – ma za zadanie wygenerowanie zoptymalizowanego kodu do pliku, na podstawie otrzymanego drzewa składniowego z modułu optymalizacyjnego

Sposób optymalizacji:

Wyciągnięcie polega na usunięciu fragmentu drzewa znajdującego się w ciele pętli i umieszczenie go w miejscu znajdujące się bezpośrednio przed pętlą.

Obliczenie może być wyciągnięte poza pętlę pod warunkiem stałości elementów operacji:

```
for (i=1; i<100; i=i + 1)
{
        b[i] = c[i] *a * 135.8;
}
```
Wyciągnięcie a * 135.8 jest możliwe;

```
for (i=1; i<100; i=i + 1)
{
        b[i] = c[i] *a * 135.8;
        a = a * 2;
}
```
a zostało nadpisane, wyciągnięcie obliczenia nie jest możliwe.

W celu określenia czy elementy operacji są stałe należy przeanalizować drzewo składniowe. Jeśli w bloku kody, w którym znajduje się rozpatrywana operacji doszło do nadpisania konkretnego elementu, to wyciągnięcie go nie jest możliwe.

Etapy optymalizacji:

1. Wyszukanie wyrażenia znajdującego się w pętli,
2. Podział wyrażenia na obliczenia według występujących operatorów logicznych
3. Kolejny podział na elementy według występujących sum i odejmowań
4. Szukanie elementów możliwych w całości do wyciągnięcia
    a. Przeszukiwanie w drzewie składniowym nadpisań występujących w elemencie zmiennych
    b. Wybranie elementów, których wszystkie czynniki mogą być wyciągnięte

    c. Określenie możliwego zasięgu wyciągnięcia wybranych elementów
    d. Określenie typu generowanej zmiennej
    e. Stworzenia nowej zmiennej i inicjalizacja przed pętlą
5. W niewyciągniętych elementach poszukiwanie możliwych do wyciągnięcia czynników(kroki jak w 4)
6. Korekcja znaków w pozostawionym wyrażeniu

Sposób testowania:

Testowanie będzie przeprowadzone w postaci testów jednostkowych i przykładów sprawdzających poprawność wykrywania błędów w kodzie oraz poprawność przeprowadzanych operacji. Poprawność optymalizacji będzie sprawdzana przez porównanie wyników przed i po wprowadzeniu zmian do kodu przez wywołanie obu wersji w kompilatorze języka C.