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
- Poprawne wykonywanie podanych instrukcji
- 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 mnożenia z pętli:

```
int main()
{
	int i, 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, 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 warunku z pętli:

```
int main()
{
	int i, j, a=4;
	for (i=1; i<100; i= i + 1)
	{
		if(a%2 == 0) j = j + 1;
	}
	return 0;
}
```

Kod po przekształceniu:

```
int main()
{
	int i, j, a=4;
	bool gen1 = a%2;
	for (i=1; i<100; i= i + 1)
	{
		if(gen1) j = j + 1;
	}
	return 0;
}
```

3. Wyciągnięcie mnożenia z zagnieżdżonej pętli:

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

Kod po przekształceniu:

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

4. Wyciągnięcie mnożenia z zagnieżdżonej pętli I główniej pętli:

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

Kod po przekształceniu:

```
int main()
{
	int i, j, a=5;
	int gen1 = a * 4;
	for (i=1; i<100; i= i + 1)
	{
		Int gen1 = a * 4;
		for (j=1; j<100; j = j + 1)
		{
			b[i] = b[i] + c[j] * 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;

Gramatyka:

Program = int main() MainBlock

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

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

ForBlock = ”{“ { IfStatement | InitStatement | AssignStatement | ForStatement | ForBlock | “break” “;” | “continue” “;” } “}” ;

ForStatement = “for” “(“ [variable AssignOp FiniteNumber]”; ”[Condition] ”; ”[AssignStatement] ”) ” ”{“ ForBlock “}”;

IfStatement = “if” “(“ Condition “)” StatementBlock [“else” StatementBlock];

InitStatement = Type (Variable | ArrayVariable) [AssignOp Assignable] “;” ;

AssignStatement = (Variable | ArrayVariable) AssignOp Assignable “;” ;

Condition = BaseCondition { LogicalOp BaseCondition};

BaseCondition = [NegationOp] Assigable RelationOp Assigable;

Assignable = Value {(AdditiveOp| MultiplicativeOp) Value}

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

Variable = Letter { Letter, Digit } ;

Index = ”[” Number ”] ” ;

ArrayVariable = Variable Index ;

Value = Variable | FiniteNumber | ArrayVariable ;

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

LogicalOp = ”|| ” | ”&& ” ;

NegationOp = “!” ;

AssignOp = “=” ;

AdditiveOp = ”+ ” | ”- ” ;

MultiplicativeOp = ”* ” | ”/ ” | ”% ” ;

Letter = ”a”…”z” | ”A”…”Z” ;

Digit = ”0”… ”9” ;

NoZero = ”1”… ”9” ;

Number = ”0” | (NoZero , {Digit});

FiniteNumber = [“-“] Number [“.” Digit {Number}]

Wymagania funkcjonalne:

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

Wymagania niefunkcjonalne:

- Program w sposób jasny wskazuje znalezione błędy
- Zmiany w kodzie powstałe w skutek optymalizacji są w wyraźny sposób pokazane użytkownikowi

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
- Główne moduły:
    - 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 dzrzewo składniowe
    - Moduł analizatora semantycznego – sprawdza poprawność utworzonego przez parser drzewa.
    - Moduł optymalizacyjny – dokonuje zmian w drzewie w celu optymalizacji w pętlach for
    - Moduł wykonawczy – ma za zadanie sekwencyjne wykonanie instrukcji zawartych w drzewie.

Sposób optymalizacji:

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

1. Wyciąganie przypisania

Przypisanie może być wyciągnięte poza pętlę pod warunkiem:

a) Wartość przypisywana nie jest uzależniona od samej siebie.
b) Wartość przypisywana nie jest uzależniona od elementu niestałego.
c) W bloku jest kolejne przypisanie, gdy jest uzależniony od siebie samego.

```
for (i=1; i<100; i=i + 1)
{
        a = b * c;
}
```

Wyciągnięcie a = b *c; jest możliwe

```
for (i=1; i<100; i=i + 1)
{
        a = a * 2;
}
```

Wyciągnięcie nie jest możliwe ->nadpisanie samej siebie

```
for (i=1; i<100; i=i + 1)
{
        a = i;
}
```

Wyciągnięcie nie jest możliwe -> element niestały

```
for (i=1; i<100; i=i + 1)
{
        a = b * c;
        b = b + 2;
}
```

Wyciągnięcie nie jest możliwe -> element niestały

```
for (i=1; i<100; i=i + 1)
{
        a = b *c;
        a = i;
}
```

Wyciągnięcie a = b *c; jest możliwe -> 2 nadpisanie jest niezależne od pierwszego

```
for (i=1; i<100; i=i + 1)
{
        a = b *c;
        a = a + i;
}
```

Wyciągnięcie nie jest możliwe -> 2 nadpisanie zależne do 1

W celu określenia czy elementy operacji są stałe należy przeanalizować drzewo składniowe. Jeśli w bloku kody, w którym znajduje się operacja przypisana została do niego nowa nowa wartość, to należy go uznać za niestały.

2. Wyciąganie części operacji

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

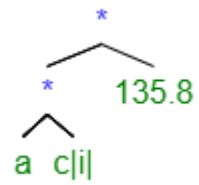
```
for (i=1; i<100; i=i + 1)
{
        b[i] = c[i] *a * 135.8;
}
```

Wyciągnięcie a * 135.8 jest możliwe;

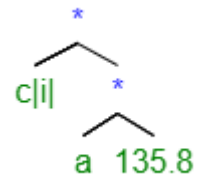
```
for (i=1; i<100; i=i + 1)
{
        b[i] = c[i] *a * 135.8;
        a = a * 2;
}
```

a zostało nadpisane, wyciągnięcie obliczenia nie jest możliwe.

W wypadku obecności więcej niż 2 elementów w operacji możliwe jest że konstrukcja drzewa nie wskazuje na możliwość wyciągnięcia obliczenia, mimo że jest to możliwe. W tym celu jeśli stwierdzimy że część elementów operacji nie podlega nadpisaniu, to należy sprawdzić czy istnieje permutacja drzewa zgodna z kolejnością działań, która pozwala na umieszczenie stałych elementów obok siebie.



a jest mnożone najpierw przez c[i] – element niestały -> wyciągnięcie niemożliwe



a jest mnożone przez 135.8 -> wyciągnięcie możliwe.

3. Pętle zagnieżdżone:

W wypadku zagnieżdżeń pętli analizę kodu należy przeprowadzić rekurencyjnie, w celu rozpoczęcia od najbardziej zagnieżdżonej pętli. W wypadku możliwości wyciągnięcia obliczenia przed daną pętle jest to wykonywane od razu, a sprawdzenie czy da się je wyciągnąć dalej jest analizowane w iteracji właściwej dla pętli do której obliczenie zostało przesunięte.

Sposób testowania:

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