Tomasz Słojewski
nr albumu 300268

TKOM dokumentacja wstępna

Zadanie O3

Zdanie polega na stworzeniu programu który będzie optymalizował kod programu w taki sposób, aby wyciągać z pętli i przenosić przed pętle te instrukcje które są możliwe do przeniesienia bez wpływania na wynik działania kodu.
Kod może zawierać zagnieżdżone pętle. Jedna instrukcja może być przeniesiona poza kilka zagnieżdżonych pętli.

Język w jakim będzie pisany kod poddawany optymalizacji:
podzbiór języka **C**

Instrukcje i wyrażenia dostępne w optymalizowanym kodzie:

- pętla for
  - for(*przypisanie_wartości_iteratorowi*; *warunek_logiczny*; *aktualizacja_iteratora*) {*zbiór_instrukcji*}
- instrukcja przypisania
  - *nazwa_zmiennej = liczba*;
  - *nazwa_zmiennej = wyrażenie*;
- wyrażenie arytmetyczne
  - *liczba operator nazwa_zmiennej*
  - *liczba1 operator liczba2*
  - *nazwa_zmiennej operator liczba*
  - *nazwa_zmiennej1 operator nazwa_zmiennej2*
- dostępne operatory arytmetyczne
  - mnożenie: *
  - dzielenie: /
  - dodawanie: +
  - odejmowanie: -
- operatory logiczne
  - mniejsze: <
  - większe: >
  - równe: ==
- inkrementacja
  - *zmienna*++;
  - ++*zmienna*;
- dekrementacja
  - *zmienna*--;
  - --*zmienna*;
- odwołanie do indeksu w tablicy (każdy indeks w tablicy traktowany jest jako zmienna)
  - *nazwa_tablicy*[*numer_indeksu*]
- inicjacja zmiennej
  - typ nazwa_zmiennej;
  - typ nazwa_zmiennej = wyrażenie;

Obsługiwane typy danych:

- int
- long
- double

Instrukcje które mogą być przenoszone:

- zmienna = liczba;
  Przykłady:
  - a = 5;
  - a = 3.14;
- zmienna1 = zmienna2;
  - a = b;
  - a = tab[0];
- zmienna = wyrażenie;
  - a = 5 * 3;
  - a = b + 4;
  - a = b * 3 + c / 5;

Gramatyka

program = {operation}

operation = loop | ([arithmeticExpression | condition | incrementation | assigment | initiation], “;”);

loop = “for(”, [initiation | assigment], “;”,condition,”;”,[incrementation | assigment],”)”,”{“, {operation},”}”

initiation = type, table | variable | assigment, “;”;

assigment = variable | table, “=”, arithmeticExpression;

condition = aritmeticExpression, relationOperator, arithmeticExpression;

arithmeticExpression = number | variable | table | incrementation, [additiveOperator | multiplicativeOperator, arithmeticExpression];

incrementation = (incrementalOperator, (variable | table)) | ((variable| table), incrementalOperator);

table = variable , “[“, arithmeticExpression, “]”;

variable = sign, {sign| digit};

number = (“0”|digitWithoutZero,{digit}),[“.”,digit,{digit}];

digit = “0” | digitWithoutZero;

digitWithoutZero = “1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”;

sign = letter | “_”;

letter = “a”|….|”z”|”A”|….|”Z”;

relationOperator = “<” | “>” | “==”;

additiveOperator = “+” | “-”;

multiplicativeOperator = “*” | “/”;

incrementalOperator = “++” | “–”;

type = “int” | “long” | “double”;

Przykłady optymalizacji kodu:

Przed optymalizacją	Po optymalizacji

int a;
int b[100];
int c[100];
for (int i=1; i<100; i++) { a=5; b[i] = c[i] *a; }
	int a;
	int b[100];
	int c[100];
	a=5;
	for (int i=1; i<100; i++) { b[i] = c[i] *a; }

int B = 10;
int a, c[100], i;
for (i=1; i<20; i++){a = B; c[i] = a * i;}
	int B = 10;
	int a, c[100], i;
	a = B;
	for (i=1; i<100; i++){c[i] = a * i;}

int i, j;
int a, b[20];
for (i=1; i<20; i++){
for (j=1; j<20; j++){a = 5; b[j] = a * i;
}}
	int i, j;
	int a, b[20];
	a = 5;
	for (i=1; i<20; i++){
	for (j=1; j<20; j++){b[j] = a * i;
	}}

int i, j;
int a, b[20], c[20];
for (i=1; i<20; i++){
for (j=1; j<20; j++){a = 5; b[j] = a * i;
}
a = 5 * b[i];
c[i] = a – 10;
}
	int i, j;
	int a, b[20], c[20];
	for (i=1; i<20; i++){
	a = 5;
	for (j=1; j<20; j++){b[j] = a * i;
	}
	a = 5 * b[i];
	c[i] = a – 10;
	}

int a, b[20]; for (int i=1; i<20; i++){a = 5; b[i] = 2 * a; a = a + 1;}	int a, b[20]; for (int i=1; i<20; i++){ a = 5; b[i] = 2 * a; a = a + 1;}

int i, a, b[20], c;
for (i=1; i<20; i++){a = 5; c = 10; b[i] = 2 * a + c; a = a + 1;}int i, a, b[20], c;
for (i=1; i<20; i++){a = 5; c = 10; b[i] = 2 * a + c; a = a + 1;}
	int i, a, b[20], c;
	c = 10;
	for (i=1; i<20; i++){a = 5; b[i] = 2 * a + c; a = a + 1;}

Wymagania:

- Odczytywanie, analizowanie i optymalizowanie podanego kodu zgodnego z zdefiniowaną składnią.
- Informowanie o liczbie zastosowanych optymalizacji.
- Zgłaszanie błędów wykrytych podczas analizy kodu
- Tworzenie zmiennych trzech obsługiwanych typów
- Wykonywanie wyrażeń arytmetycznych uwzględniając kolejność operatorów.
- Tworzenie zmiennych.
- Obsługa zakresu dostępności zmiennych.
- Używanie pętli for
- Kod po optymalizacji musi być równoznaczny z kodem oryginalnym.
- Maksymalna długość pojedynczego tokena będzie wynosiła 64 znaki.
- Sygnalizowanie próby pobrania kodu z nieistniejącego pliku lub z takiego do którego nie ma dostępu.

Błędy:

Program będzie obsługiwał następujące błędy:

- Niezdefiniowany token – ten błąd nie będzie powodował przerwania analizy leksykalnej kodu
- Zbyt długa nazwa zmiennej
- Nieznana nazwa
- Oczekiwano nazwy zmiennej
- Brakujący średnik
- Brak „]” (przy indeksowaniu tablicy)

Działanie programu:

Program zostanie napisany przy użyciu języka C++.
Plik z kodem przeznaczony do optymalizacji będzie przekazywany jako argument programu. Do projektu będzie dołączony plik makefile który będzie umożliwiał kompilację przy pomocy polecenia *make*. Uruchamianie programu będzie dokonywane przy pomocy komendy
./optymalizator plik_wejściowy [plik_wyjściowy]
Kod po optymalizacji będzie umieszczany w pliku wyjściowym o podanej nazwie lub o nazwie domyślnej.

Testowanie będzie odbywało się przy pomocy testów jednostkowych z biblioteki boost.
Do stworzenia testów jednostkowych wykorzystane zostaną wszystkie przykłady podane w tabeli wyżej. Testowanie będzie polegało na usunięciu wszystkich białych znaków z kodu, a następnie porównaniu otrzymanego kodu z oczekiwanym.

Analiza kodu

Analiza kodu będzie się składała z 3 poziomów. Każdy z poniższych poziomów będzie działał równolegle, używając wielowątkowości.

Analizator leksykalny:

Podzieli kod na tokeny. Będzie wczytywał kolejne znaki, aż do stwierdzenia końca tokenu. Każdy z tokenów będzie reprezentowany przez obiekt klasy Token.
Każdy z tokenów będzie posiadał dwa pola: typ tokenu oraz wartość będącą typem całkowitym, zmiennopozycyjnym lub tekstowym.
Tokeny będą następujących typów:

- Słowo kluczowe
- Operator przypisania
- Operator addytywny
- Operator multiplikatywny
- Operator inkrementacyjny/dekrementacyny
- Typ danych
- Operator relacyjny
- Nawias okrągły
- Nawias blokowy
- Liczba
- Koniec pliku

Analizator składniowy:

Będzie pobierał od analizatora kolejne Tokeny, a następnie grupował je w drzewo, jednocześnie sprawdzając, poprawność gramatyczną kodu, zgodną z gramatyką podaną powyżej.

Analizator semantyczny:

Będzie pobierał od analizatora składniowego drzewa i analizował je pod względem poprawności semantycznej. Analizator będzie przechodził po drzewie w głąb.
Analizator semantyczny będzie też sprawdzał istnienie i zasięg zdefiniowanych zmiennych, łącząc się z tablicą symboli.

Po trzech etapach analizowania kodu nastąpi jego optymalizacja. Będzie ona polegała na analizę drzewa w i dla każdej ze zmiennych sprawdzaniu czy dana zmienna została edytowana.

Tablica symboli:

Tablica symboli będzie mapą, której kluczami będą stringi oznaczające nazwę zmiennej, a wartością będzie struktura zawierająca typ i wartość zmiennej. Zmienne tablicowe będą przechowywane jako string w postaci: ”zmienna[index]”