

Лабораторна робота №2.

Трансляція мов високого рівня у мови низького рівня. Частина 2

Мета лабораторної роботи наступна:

- ознайомитись із роботою сучасних трансляторів на прикладі трансляції невеликої програми, що містить різні конструкції керування потоком виконання і написаної мовою програмування високого рівня, у код цієї програми мовою низького рівня;
- навчитись виокремлювати синтаксичні/семантичні конструкції програми, записаної мовою високого рівня, у відповідній їй програмі, записаній мовою низького рівня, на прикладі мови високого рівня Java.

1. Вимоги до апаратного і програмного забезпечення

Для виконання лабораторної роботи діють наступні вимоги щодо програмного та апаратного забезпечення:

- 1) компілятор мови Java 21 — openjdk 21 або adoptopenjdk 21.

2. Загальне завдання та термін виконання

Завдання лабораторної роботи складається з чотирьох пунктів:

1. Реалізувати програму сортування масиву згідно із варіантом мовою Java (інформація про варіанти наведена в п. 4). Програма, записана мовою Java, має бути синтаксично і семантично відповідна програмі, записаній мовою C (це досягається завдяки синтаксичній та семантичній подібності цих двох мов для випадку даного завдання), що була реалізована в лабораторній роботі №1. *Результатом* виконання цього пункту є лістинг програми мовою Java.
2. Виконати трансляцію програми, написаної мовою Java, у байт-код Java за допомогою **javac** і **java** (програми, що постачаються разом з пакетом openjdk) й встановити семантичну відповідність між командами мови Java та командами одержаного байт-коду Java шляхом додавання коментарів з поясненням. *Результатом* виконання даного пункту буде лістинг байт-коду програми із коментарями в коді, в яких наведено відповідний код програми, записаної мовою Java.
3. Розібратись і вміти пояснити, що виконують ті чи інші команди байт-коду Java, що будуть присутні в байт-коді програми, отриманого в другому пункті загального завдання; вміти пояснити зв'язок між кодом мовою Java та байт-кодом.

4. Виконати порівняльний аналіз відповідних семантичних частин програм, записаних мовою асемблера (лабораторна робота №1) та байт-кодом. *Результатом* виконання даного пункту буде таблиця із порівнянням відповідних частин асемблерного коду та байт-коду.

В результаті виконання лабораторної роботи має бути підготовлено **звіт**, в якому будуть зазначені наступні пункти:

- 1) титульний аркуш;
- 2) загальне завдання лабораторної роботи;
- 3) варіант і завдання за варіантом до першого пункту загального завдання (інформація про варіанти наведена в п. 4);
- 4) лістинг програми мовою Java;
- 5) лістинг байт-коду програми;
- 6) порівняльна таблиця.

Граничний **термін виконання** лабораторної роботи визначається викладачем в інформаційному листі, що надсилається на пошту групи та/або в групу в телеграмі разом із даними методичними вказівками. В разі недотримання граничного терміну виконання лабораторної роботи, максимальна оцінка за роботу буде знижена на 1 бал за кожен тиждень протермінування. Кількість балів за дотримання дедлайну даної лабораторної роботи визначається силабусом (PCO). Ця ж кількість відповідає максимальній кількості балів, на яку може бути знижена оцінка за лабораторну роботу в разі недотримання дедлайну.

3. Методичні вказівки

В даному розділі наведено методичні вказівки щодо виконання пунктів завдання лабораторної роботи на спрощеному прикладі.

Нехай маємо наступне завдання: реалізувати програму (функцію) визначення суми парних чисел в масиві заданої довжини.

3.1 Реалізація завдання за варіантом мовою Java

Реалізацією поставленого завдання мовою Java буде наступний код:

```
package test;

public class Main {
    public static void main(String[] args) {
        int arr[] = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        int sum = calculateEvenSum(arr, arr.length);
        System.out.printf("Sum is: %d\n", sum);
        return;
    }
}
```

```

private static int calculateEvenSum(int[] arr, int size) {
    int sum = 0;
    // main loop
    for (int idx = 0; idx < size; idx++) {
        if ((arr[idx] & 1) == 0) {
            sum += arr[idx];
        }
    }
    return sum;
}
}

```

Вимогою до реалізації завдання за варіантом є максимальне збереження подібності коду, написаного мовою Java, коду, написаному мовою C (лабораторна робота №1).

Варто зауважити, що типи даних в мові C (лабораторна робота №1) та Java дещо відрізняються. Наприклад, розмір масиву в C передається в якості аргументу функції. В Java масив — це об'єкт, у якого є *final* поле (*length*), яке зберігає довжину масиву. І хоча дізнатись довжину масиву можна напряду з об'єкту масиву, в прикладі розв'язання задачі вище розмір було передано в якості параметра функції. Це зроблено з метою забезпечення більшої подібності між кодом мовою C та Java.

Як і у випадку реалізації завдання мовою C, для даного прикладу у звіт має потрапити лише код методу *calculateEvenSum*.

При реалізації програми мовою Java слід дотримуватись конвенцій програмування мовою Java. Наприклад, [Java Code Conventions](#).

3.2 Трансляція коду мовою Java в байт-код Java

Отримати байт-код програми мовою Java можна у два етапи: компіляція і дизасемблювання.

Спочатку необхідно скомпілювати *.java* файл і отримати з нього *.class* файл наступною командою:

```
javac <FileName.java>
```

<FileName.java> - ім'я файлу з реалізацією завдання.

Якщо в коді мовою Java будуть помилки — *javac* виведе повідомлення про них. Після того, як всі помилки будуть виправлені, на виході буде отримано файл з розширенням *.class*.

Після цього необхідно виконати наступну команду:

```
java -p [-p] -c <FileName.class>
```

Команда виконає дизасемблювання файлу *<FileName.class>* й виведе байт-код в термінал (або у файл, якщо потік виводу буде перенаправлено). Опис опцій:

- *-c* вказує, що необхідно виконати дизасемблювання до рівня байт-коду;

- -p вказує, що необхідно дизасемблювати всі функції (будь-якої області видимості). Цей параметр може знадобитись у випадку, якщо реалізація завдання знаходиться в окремій приватній функції.

Для функції *calculateEvenSum* було згенеровано наступний байт-код:

```
private static int calculateEvenSum(int[], int);
Code:
  0: iconst_0
  1: istore_2
  2: iconst_0
  3: istore_3
  4: iload_3
  5: iload_1
  6: if_icmpge      29
  9: aload_0
 10: iload_3
 11: iaload
 12: iconst_1
 13: iand
 14: ifne          23
 17: iload_2
 18: aload_0
 19: iload_3
 20: iaload
 21: iadd
 22: istore_2
 23: iinc          3, 1
 26: goto          4
 29: iload_2
 30: ireturn
```

Скориставшись [специфікацією мови Java](#) (21 версії), можна визначити, що роблять окремі команди байт-коду. Деякі з них наведено в таблиці 1. *Java Virtual Machine* (JVM), що лежить в основі функціонування платформи Java, опрацьовує байт-коди, виконуючи відповідні машинні інструкції на конкретній архітектурі. JVM базується на роботі зі стеком, тому більшість операцій беруть свої операнди з голови *стеку операндів*, а результат кладуть назад у стек.

Для того, щоб швидше розібратись з тим, що відбувається в байт-коді й визначити семантичні відповідники між програмою, записаною мовою Java, та програмою, записаною байт-кодом Java, можна скористатись наступними сайтами: [Compiler Explorer](#) (рис. 1) або [Byte Me](#) (рис. 2).

Варто також зазначити, що локальними змінними 0 та 1 в байт-коді функції *calculateEvenSum* є формальні параметри *arr* та *size*.

Після аналізу семантичних відповідників, необхідно додати коментарі до байт-коду, які пояснюють що в ньому відбувається. В протоколі такі позначення необхідно виділити кольором. Зокрема, необхідно додати такі позначення:

- початки (позначаються перед першою інструкцією байт-коду) і кінці (позначаються після останньої інструкції байт-коду) структурних блоків:
 - тіло функції;

- цикл;
- тіло циклу;
- умова циклу;
- умова умовного оператора if/switch;
- true/false гілки умовного оператора if;
- case/default гілки оператора розгалуження switch;
- відповідні фрагменти коду мовою Java біля останньої інструкції байт-коду, що їх реалізує.

Таблиця 1. Опис деяких команд байт-коду Java

Команда байт-коду	Опис
<code>iconst_0</code>	вштовхнути ціле число 0 до стеку операндів
<code>istore_2</code>	зберегти операнд з голови стеку операндів до локальної змінної 2
<code>iload_1</code>	вштовхнути цілочисельне значення локальної змінної 1 до стеку операндів
<code>if_icmpge 29</code>	порівняти два числа зі стеку операндів й перейти на мітку 29, у випадку, якщо перше число більше або рівне другому
<code>aload_0</code>	вштовхнути значення локальної змінної 0 (значення є посиланням) до стеку операндів
<code>iand</code>	булева операція “І” над двома цілими числами
<code>goto 4</code>	перехід на мітку 4
<code>iinc 3, 1</code>	збільшення значення цілочисельної локальної змінної 3 на константу 1
<code>ireturn</code>	повернення цілого числа з голови стеку операндів з методу

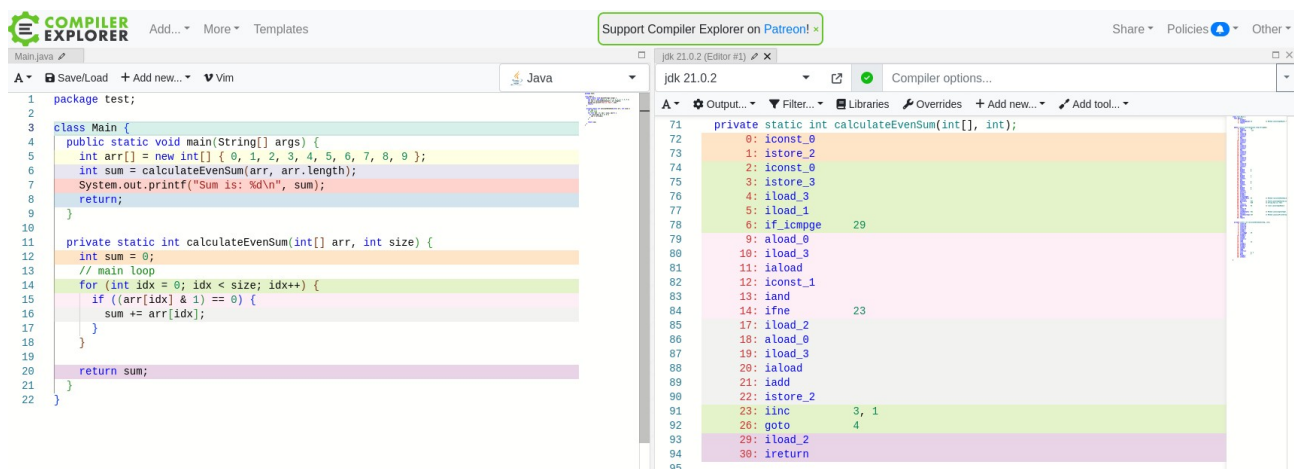



Рис. 1. Використання Compiler Explorer для визначення семантичних відповідників



Byte-Me: A Java Bytecode Explorer

Edit Source

Load Source

Upload Class

Core Libs

Compiled with -source 21 -target 21

Class: test.Main

Member: calculateEvenSum([I])

Constant Pool ☒

Show Details ☒

Java Source Code		Bytecode		Constant Pool		Class Info	
Line	Instructions	BCI	Instruction	Operands			
1	package test;						
2							
3	public class Main {						
4	public static void main(String[] args) {						
5	int arr[] = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };	0	iconst_0		16	java/lang/System	Utf8
6	int sum = calculateEvenSum(arr, arr.length);	1	istore_2		17	out	Utf8
7	System.out.printf("Sum is: %d\n", sum);	2	iconst_0		18	Ljava/io/PrintStream;	Utf8
8	return;	3	istore_3		19	Sum is: %d	String
9	}				20	Sum is: %d	Utf8
10					21	java/lang/Integer.valueOf(I)Ljava/lang/Integer;	MethodRef
11	private static int calculateEvenSum(int[] arr, int size) {	4	iload_3		22	java/lang/Integer	Class
12	int sum = 0;	5	iload_1		23	valueOf(I)Ljava/lang/Integer;	NameAndType
13	// main loop	6	if_icmpge	bci 29	24	java/lang/Integer	Utf8
14	for (int idx = 0; idx < size; idx++) {	7	aload_0		25	valueOf	Utf8
15	if ((arr[idx] & 1) == 0) {	10	iload_3		26	(I)Ljava/lang/Integer;	Utf8
16	sum += arr[idx];	11	iaload		27	java/io/PrintStream.printf(Ljava/lang/String;(Ljava/lang/Object;)Ljava/io/PrintStream;	MethodRef
17	}	12	iconst_1		28	java/io/PrintStream	Class
18	}	13	iand		29	printf(Ljava/lang/String;(Ljava/lang/Object;)Ljava/io/PrintStream;	NameAndType
19		14	ifne	bci 23	30	java/io/PrintStream	Utf8
20		17	iload_2		31	printf	Utf8
21		18	aload_0		32	(Ljava/lang/String;(Ljava/lang/Object;)Ljava/io/PrintStream;	Utf8
22		19	iload_3		33	Code	Utf8
		20	iaload		34	LineNumberTable	Utf8
		21	iadd		35	main	Utf8
		22	istore_2		36	((Ljava/lang/String;)V	Utf8
		23	iinc	local slot 3 immediate 1	37	StackMapTable	Utf8
		26	goto	bci 4	38	SourceFile	Utf8
					39	Main.java	Utf8

Class Info	
Major Version	65 (Java 21)
Minor Version	0
Magic Number	CAFEBABE
Access Flags	PUBLIC, SUPER
Attributes	SourceFile
Fields	
Methods	<init> main calculateEvenSum
Interfaces	
Superclass	java/lang/Object
Permitted subclasses	

Method Info	
Name	calculateEvenSum
Access	PRIVATE, STATIC
Attributes	Code
Exceptions	

Code Info	
Max Stack	3
Max Locals	4
Attributes	LineNumberTable

Рис. 2. Використання Byte-Me для дослідження байт-коду Java

Після аналізу байт-коду, необхідно додати коментарі, які пояснюють що відбувається в коді:

```

private static int calculateEvenSum(int[], int);
Code:
// method body start
0: iconst_0          // load constant 0 to stack
1: istore_2          // int sum = 0;
// main loop start: for (int idx = 0; idx < size; idx++)
2: iconst_0          // load constant 0 to stack
3: istore_3          // initialize idx: int idx = 0
// main loop exit condition start
4: iload_3           // load int value of local variable idx to stack
5: iload_1           // load int value of formal variable size to stack
6: if_icmpge         29 // check main loop continuation condition: idx < size
                        // and jump to label 29 when if fails
// main loop exit condition end
// main loop body start
// if check start: if ((arr[idx] & 1) == 0)
9: aload_0           // load value (which is reference) of formal variable
                        // arr to stack
10: iload_3           // load int value of local variable idx to stack
11: iaload            // load int value from array (arr[idx]) to stack
12: iconst_1         // load constant 1 to stack
13: iand              // perform bitwise AND operation on int operands:
                        // arr[idx] & 1
14: ifne              23 // check if value in stack is equal to 0:
                        // (arr[idx] & 1) == 0. If it is not – jump to label
                        // 23:
// if check end
// if true branch start
17: iload_2           // load int value of local variable sum to stack
18: aload_0           // load value (which is reference) of formal variable
                        // arr to stack
19: iload_3           // load int value of local variable idx to stack
20: iaload            // load int value from array (arr[idx]) to stack

```

```

21: iadd          // perform operation ADD over to ints: sum + a[idx]
22: istore_2      // save int result of ADD to sum: sum += arr[idx]
// if true branch end
// main loop body end
23: iinc          3, 1 // increment idx: idx++
26: goto         4     // go to loop exit condition
// main loop end
29: iload_2       // load value saved in sum to operand stack
30: ireturn      // return sum from method: return sum;
// function body end

```

Байт-код з коментарями і має потрапити до звіту. Коментарі, позначені синім та зеленим кольорами у прикладі, мають обов’язково бути присутніми у звіті (у звіті їх так само варто виділити кольором). Частини асемблерного коду, відмічені коментарями у прикладі (маються на увазі всі коментарі “// ...”, а не лише виділені кольором), необхідно вміти пояснити. При цьому коментарі з поясненням можна або додати до звіту, або не додавати (але позначені кольором коментарі мають бути обов’язково). Синім позначені коментарі, що виділяють окремі логічні частини програми (початок певного циклу, перевірка умови циклу, початок тіла циклу, перевірка умови певного if, true/else гілки if та ін.), а зеленим — позначаються коментарі з відповідниками коду мовою Java (наприклад, інструкції iadd відповідає операція додавання + в програмі, записаній мовою Java). Деякі відповідники не є точними, оскільки одна інструкція байт-коду не завжди відповідає одній інструкції мови Java. Наприклад, це стосується операцій присвоєння. Присвоєння 0 у змінну sum відбувається за допомогою двох інструкцій байт-коду: iconst_0 та istore_2.

3.3 Порівняльний аналіз асемблерного коду і байт-коду Java

Порівняння асемблерного коду і байт-коду Java наведено в таблиці 2.

В результаті в таблиці має бути зазначено всі унікальні відповідники, які є в реалізованій програмі. Повторення (наприклад, визначення різних локальних змінних і встановлення в них певних значень) можна не додавати. Окрім простих виразів (операції присвоєння, інкременту, додавання і т. і.) в таблиці також має бути присутнє порівняння (і аналіз) конструкцій, що керують потоком виконання (if, for, та ін.).

4. Варіанти завдань

В якості завдань необхідно використати завдання з лабораторної роботи №2.2 з курсу “Структури даних та алгоритми”, що викладався на першому курсі. Конкретний варіант обирається такий самий, як і для лабораторної роботи №1 курсу Архітектура комп’ютерів. Архітектура для програмістів.

5. Контрольні питання

- 1) Що таке закон Мура?
- 2) Опишіть закон Деннарда.

- 3) Основні визначення стосовно архітектури комп'ютерів.
- 4) Принципи побудови комп'ютерів.
- 5) Поясніть, що виконують команди байт-коду Java (X, Y — змінювані): Xload, Xfne, goto, Xstore_Y, Xaload та інші.

Таблиця 2. Семантичні відповідники між асемблерним кодом і байт-кодом Java

№	Код мовою C	Код мовою Java	Асемблерний код	Байт-код Java	Опис
1	int sum = 0;	int sum = 0;	mov DWORD PTR -8[rbp], 0	iconst_0 istore_2	визначення змінної sum і запис в неї значення 0. Через необхідність роботи з неявними параметрами, що беруться зі стеку, байт-код налічує дві інструкції для аналогічного коду, записаного мовою асемблера
2	arr[idx]	arr[idx]	mov eax, DWORD PTR -4[rbp] cdqe lea rdx, 0[0+rax*4] mov rax, QWORD PTR -24[rbp] add rax, rdx mov eax, DWORD PTR [rax]	aload_0 iload_3 iaload	отримання значення з визначеної комірки масиву (за індексом). В асемблерному коді відбувається приведення типів даних, і вирахування адреси, через що кількість команд більша, ніж в байт-коді
3	idx < size	idx < size	mov eax, DWORD PTR -4[rbp] cmp eax, DWORD PTR -28[rbp] jl .L4	iload_3 iload_1 if_icmpge 29	перевірка умови виходу з циклу
4	arr[idx] & 1	arr[idx] & 1	<#2> and eax, 1	<#2> iconst_1 iand	<#2> - див. рядок 2. виконання операції “I”. Операнд ‘1’ операції and в асемблерному коді вказується прямо в команді, а в байт-коді передається через стек операндів
5	(arr[idx] & 1) == 0	(arr[idx] & 1) == 0	<#2> <#4> test eax, eax jne .L3	<#2> <#4> ifne 23	перевірка умови в if. Мовою асемблера операція порівняння виконується двома інструкціями (встановлення прапорців і умовний перехід), в той час як байт-код містить лише одну команду, що виконує порівняння і перехід

6	if (<cond>) <statement> ...	if (<cond>) <statement> ...	<cond> jne .L3 <statement> .L3 ...	<cond> ifne 23 <statement> 23: ...	реалізація умовного переходу if
7	for (int idx = 0; idx < size; idx++) <statement>	for (int idx = 0; idx < size; idx++) <statement>	... (todo)	... (todo)	реалізація циклу з лічильником for. Відмінність полягає в тому, що ... (todo)
...