МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИиППО)

ОТЧЁТ ПО ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

по дисциплине

«Системное программное обеспечение»

Выполнил студент группы ИКБО-03-18 Маковецкий И. А.

Принял Соболев О. В.

Практические занятия выполнены «____» _____ 2021 г. (подпись студента)

Практические занятия зачтены «____» _____ 2021 г. (подпись руководителя)

Москва 2021

СОДЕРЖАНИЕ

1 ПРАКТИЧЕСКАЯ РАБОТА № 1

1.1 Тема

Генерация кода для выражения с константами и переменными.

1.2 Задание

Реализовать генерацию ассемблерного кода для арифметического выражения из констант и переменных.

1.3 Ход выполнения работы

В ходе выполнения работы был реализован модуль транслятора, осуществляющий генерацию asm-кода для арифметических выражений, состоящих из целочисленных констант и операций сложения, вычитания, умножения, целочисленного деления, взятия остатка от деления. Разработка опирается на лексический и синтаксический анализатор, реализованные в результате освоения дисциплины «Теория автоматов и формальных языков». Входными данными для модуля является абстрактное синтаксическое дерево (AST), получаемое в результате лексического и синтаксического разбора математических выражений, записанных в текстовом виде. Выходными данными является asm-код, представляющий собой код вычисления выражения, поданного в качестве входных данных.

Генерация кода для узлов AST выполняется по правилам, описанным ниже. Для узлов, представляющих константу — команда ассемблера $PUSH$

$QWORD < n >$, где $< n >$ — константа, хранящаяся в узле. Для узлов с бинарной операцией — команды ассемблера:

- $POP\ RBX$; второй операнд бинарной операции — сверху стека;
- $POP\ RAX$; первый операнд бинарной операции — под вторым;
- Код конкретной операции;
- $PUSH\ RAX$; помещение результата операции на вершину стека.

Ниже приведен фрагмент кода обхода AST.

```
public static void generateASM(Set<String> vars) {
    for (String var: vars) {
        System.out.println("MOV RCX, promt_" +  var);
        System.out.println("MOV R11, printf");
        System.out.println("CALL R11");

        System.out.println(" ");
        System.out.println("MOV RDX, scanf_format");
        System.out.println("MOV RDX, " + var);
        System.out.println("MOV R11, scanf");
        System.out.println("CALL R11\n");
    }
}

public static void main(String[] args) {
    String text = "x + y + 2";

    Lexer l = new Lexer(text);
    List<Token> tokens = l.lex();
    tokens.removeIf(t -> t.type == TokenType.SPACE);
```

```
        Parser p = new Parser(tokens);
        ExprNode node = p.parseExpression();
        Set<String> vars = new LinkedHashSet<>();
        getVars(vars, node);

        System.out.println("section .text\n" +
            " global main\n" +
            " extern printf\n" +
            " extern scanf\n" +
            " \n" +
            "main: ");
    }
```

1.4 Вывод

В ходе проделанной работы было проведено ознакомление с ассемблером для архитектуры Intel x86 — nasm, а также его применением для написания простейших программ для вычисления значения арифметических выражений. В результате выполнения работы реализована генерация ассемблерного кода для вычисления значения арифметических выражений, содержащих целочисленные константы, а также операции сложения, вычитания, умножения, целочисленного деления, взятия остатка от деления.

Полный исходный код реализованной программы доступен по адресу https://github.com/ivanmakovetskiy/spo/blob/master/spo-main/lang/Compiler.java.

2 ПРАКТИЧЕСКАЯ РАБОТА № 2

2.1 Тема

Оптимизация кода (свертка констант и $peephole\ optimization$).

2.2 Задание

Реализовать оптимизацию ассемблерного кода для арифметического выражения из констант и переменных.

2.3 Ход выполнения работы

Реализуем две оптимизации для транслятора, созданного в рамках выполнения предыдущих работ. На уровне AST реализуем так называемую «свертку констант». Свёртка констант— оптимизация, вычисляющая константные выражения на этапе компиляции. Прежде всего, упрощаются константные выражения, содержащие числовые литералы. Для этого реализуем функцию $Expr.fold()$, которая будет обходить выражение и выполнять свертку путем замены узлов AST. Наибольший интерес в этом случае будет представлять свертка унарных и бинарных операций. Для унарных операций будем сначала выполнять свертку операнда, затем, если операнд является константой, будем применять заданную унарную операцию к этой константе и заменять узел унарной операции на узел с константой-результатом свертки. Для бинарных операций будем сначала выполнять свертку левого и правого операнда, затем, если левый и правый операнды являются константами, будем применять к ним заданную бинарную

операцию и заменять узел бинарной операции на узел с константой-результатом свертки.

На уровне генерируемого ассемблерного кода будем выполнять $peephole$-оптимизацию. Локальные $peephole$-оптимизации рассматривают несколько соседних (в терминах одного из графов представления программы) инструкций (как будто «смотрит в глазок» на код), чтобы увидеть, можно ли с ними произвести какую-либо трансформацию с точки зрения цели оптимизации. В частности, они могут быть заменены одной инструкцией или более короткой последовательностью инструкций. Будем искать в ассемблерном коде паттерны, которые можно упростить. В генерируемом созданным транслятором коде наиболее часто встречаются следующие паттерны:

```
push qword [x]
pop rbx
```

или

```
    push qword <const>
    pop rbx
```

или

```
    push rax
    pop rbx
```

Их можно заменить на

```
mov rbx, qword [x]
mov rbx, qword <const>
mov rbx, rax
```

Для реализации данной оптимизации нам потребуется хранить генерируемый ассемблерный код. В данном случае будем просто хранить инструкции в

виде списка строк. Для этого реализуем класс $InstructionBuffer$. В его экземпляр будут попадать все инструкции, генерируемые транслятором, и в нем же будет осуществляться оптимизация. Основной код оптимизации содержится в методе $performPeepholeOptimization$, вызываемом после каждого добавления новой инструкции в конец буфера. Фрагмент кода буфера, отвечающий за оптимизацию приведен ниже.

Функция $peepholeOptimize$

```
static void peepholeOptimize() {
    Pattern push = Pattern.compile("\\s*PUSH (.+)");
    Pattern pop = Pattern.compile("\\s*POP (.+)");
    for(int i = 0; i < asm.size() - 1; i++) {
        String instr = asm.get(i);
        String nextInstr = asm.get(i + 1);

        Matcher m1 = push.matcher(instr);
        Matcher m2 = pop.matcher(nextInstr);
        if (m1.matches() && m2.matches()) {
            String arg1 = m1.group(1);
            String arg2 = m2.group(1);
            asm.set(i, "    MOV " + arg2 + ", " + arg1);
            asm.remove(i + 1);
        }
    }
}
```

Функция $foldConstants$

```
public static ExprNode foldConstants(ExprNode node) {
    if (node instanceof VarNode) {
        return node;
    }
    else if (node instanceof NumberNode) {
        return node;
    }
    else if (node instanceof BinOpNode) {
        BinOpNode binOp = (BinOpNode) node;
        ExprNode l = foldConstants(((BinOpNode) node).left);
        ExprNode r = foldConstants(((BinOpNode) node).right);
        if (l instanceof NumberNode && r instanceof  NumberNode) {

        int lvalue = Integer.parseInt(((NumberNode) l).number.text);
        int rvalue = Integer.parseInt(((NumberNode) r).number.text);
        int result;
        switch (binOp.op.type) {
            case ADD:
                result = lvalue + rvalue;
                break;
            case SUB:
                result = lvalue - rvalue;
                break;
            case MUL:
                result = lvalue * rvalue;
                break;
            case DIV:
                result = lvalue / rvalue;
                break;
```

```
            default:
            throw new IllegalStateException("Unexpected value: "
                + binOp.op.type);
        }
        return new NumberNode(new Token(NUMBER,
            Integer.toString(result), binOp.op.pos));
    } else {
        return new BinOpNode(binOp.op, l, r);
    }
    }
    throw new IllegalStateException();
}
```

2.4 Вывод

В ходе проделанной работы было проведено ознакомление с простейшими оптимизациями, реализуемыми при разработке компиляторов и трансляторов. В результате выполнения работы созданный ранее транслятор дополнен двумя оптимизациями: сверткой констант на уровне AST и $peephole$-оптимизацией на уровне генерируемого ассемблерного кода. Полный исходный код реализованной программы доступен по адресу https://github.com/ivanmakovetskiy/spo/blob/master/spo-main/lang/CompilerOptimize.java

3 ПРАКТИЧЕСКАЯ РАБОТА № 3

3.1 Тема

Вывод символов объектного файла $COFF$.

3.2 Задание

Реализовать вывод символов, содержащихся в объектном файле формата $COFF$.

3.3 Ход выполнения работы

Объектный файл состоит из секций, каждая секция может содержать несколько символов. Обычно используемые секции:

- $.text$ содержит машинный код программы, для нее символы — это имена функций, код которых содержится в секции;
- $.data$ содержит инициализированные глобальные переменные, для нее символы — это имена переменных;
- $.bss$ содержит неинициализированные глобальные переменные, для нее символы — это имена переменных.

Основные элементы структуры объектного файла формата $COFF$ для $x86$ приведены в таблице 3.1.

Для того, чтобы вывести имена символов, содержащихся в объектном файле, необходимо:

Таблица 3.1: Элементы структуры объектного файла.

Элемент	Назначение	Размещение	Размер
File Header	Содержит основную информацию о файле и указатели на другие части файла.	В начале файла.	Фиксированная длина (20 байт).
Optional Header	Содержит дополнительную информацию о файле.	Следует за File Header или может отсутствовать (в таком случае его размер равен 0).	Указывается в заголовке файла.
Section Header	Содержит информацию о различных секциях, содержащихся в файле.	Сразу за File Header или Optional Header.	Равен количеству секций (указанному в заголовке) умноженному на размер одной структуры.
Symbol Table	Содержит информацию о каждом символе, объявленном или определенном в коде.	Начинается со смещений, указанного в File Header.	Количество символов, умноженное на размер одного символа.
String Table	Содержит имена символов или секций, длина которых больше 8 символов.	Непосредственно за SymbolTable.	Указывается первыми 32 битами таблицы.

- Прочитать из Заголовка файла поля «Количество символов» и «Смещение таблицы символов»;
- Загрузить таблицу имен: перейти в файле на позицию «Смещение таблицы символов» + 18*«Количество символов», прочитать 4 байта размера Таблицы имен, и загрузить это количество байт минус 4 (т.к. размер таблицы включает в себя и 4 байта размера);
- Прочитать из файла «Количество символов» записей по 40 байт, начиная с байта «Смещение таблицы символов». Для каждой записи, если она является основной записью, необходимо прочитать имя символа (непосредственно из записи, если имя занимает менее 8 символов, либо из таблицы имен в противном случае).

Код программы:

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.util.HashMap;
import java.util.Map;

public class Compiler {
```

```
public static void main(String[] args) throws IOException {
    RandomAccessFile f = new RandomAccessFile("D:/SPE3pr.obj", "r");
    byte[] header = new byte[20];
    f.read(header);
    int ns = frun16(header[2], header[3]);
    System.out.println(ns);
    int optHeader = frun16(header[16], header[17]);
    System.out.println(optHeader);
    int symbolOffset = frun32(header[8], header[9], header[10], header[11]);
    System.out.println(symbolOffset);
    f.seek(symbolOffset);
    int nSymbols = frun32(header[12], header[13], header[14], header[15]);
    System.out.println(nSymbols);
    int stringTablePosition = symbolOffset + 18 * nSymbols;
    f.seek(stringTablePosition);
    byte[] length = new byte[4];
    f.read(length);
    int stringTableLength = frun32(length[0], length[1], length[2], length[3]);
    byte[] stringTable = new byte[stringTableLength];
    f.seek(stringTablePosition);
    f.read(stringTable);
    f.seek(symbolOffset);
    for (int i = 0; i < nSymbols;) {
        byte[] sym = new byte[18];
        f.read(sym);
        String info = getName(sym, stringTable);
        System.out.println(info);
        int skip = sym[17] & 0xFF;
        if (skip > 0) {
            i += skip + 1;
            f.skipBytes(18 * skip);
        } else {
            i++;
        }
    }
```

```
    f.seek(20 + optHeader);
    Map<String, int[]> sections = new HashMap<String, int[]>();
    for(int i = 0; i < ns; i++) {
        byte[] sectTable = new byte[40];
        f.read(sectTable);
        String sectName = new String(sectTable, 0, 8, "US-ASCII");
        int lengthSect = frun32(sectTable[16], sectTable[17],
            sectTable[18], sectTable[19]);
        int offsetSect = frun32(sectTable[20], sectTable[21],
            sectTable[22], sectTable[23]);

        System.out.println(sectName);
        sections.put(sectName, new int[] {lengthSect, offsetSect});
    }

    for (Map.Entry<String, int[]> entry : sections.entrySet()) {
        System.out.println(entry.getKey());
        int[] pair = entry.getValue();
        f.seek(pair[1]);
        byte[] data = new byte[pair[0]];
        f.read(data);
        System.out.println(new BigInteger(data).toString(16));
    }

}

public static int frun16(byte b0, byte b1){
    return ((b1 & 0xFF) << 8 | (b0 & 0xFF));
}

public static int frun32(byte b0, byte b1, byte b2, byte b3){
    int i = b3 & 0xFF;
    i = (i << 8) | (b2 & 0xFF);
    i = (i << 8) | (b1 & 0xFF);
    i = (i << 8) | (b0 & 0xFF);
```

```
    return i;
}

public static String getName(byte[] data, byte[] stringTable) throws UnsupportedEnc
    int name = frun32(data[0], data[1], data[2], data[3]);
    if (name == 0) {
        int sofs = frun32(data[4], data[5], data[6], data[7]);
        int length = 0;
        for (int i = sofs; i < stringTable.length; i++) {
            if (stringTable[i] == 0) {
                break;
            }
            length++;
        }
        return new String(stringTable, sofs, length, "US-ASCII");
    }
    return new String(data, 0, 8, "US-ASCII");

    }
}
```

3.4 Вывод

В ходе проделанной работы было проведено ознакомление со структурой объектных файлов формата COFF. В результате выполнения работы реализована утилита, предназначенная для вывода символов, определенных в объектных файлах. Полный исходный код программы доступен по адресу https://github.com/ivanmakovetskiy/spo/blob/master/spo-main/Compiler.java.

4 ПРАКТИЧЕСКАЯ РАБОТА № 4

4.1 Тема

Использование системных вызовов ОС

4.2 Задание

Реализовать код с использованием языка ассемблера, используя системные вызовы.

4.3 Ход выполнения работы

Системный вызов — способ обращения программы пользовательского пространства к пространству ядра. Со стороны это может выглядеть как вызов обычной функции со своим собственным $calling\ convention$, но на самом деле процессором выполняется чуть больше действий, чем при вызове функции инструкцией $call$. Например, в архитектуре $x86$ во время системного вызова как минимум происходит увеличение уровня привилегий, замена пользовательских сегментов на сегменты ядра и установка регистра IP на обработчик системного вызова.

Программист обычно не работает с системными вызовами напрямую, так как системные вызовы обернуты в функции и скрыты в различных библиотеках, например, $libc.so$ в Linux или же $ntdll.dll$ в Windows, с которыми и взаимодействует прикладной разработчик.

Необходимо написать две функции: функцию $print$, которая выводит на

экран принимаемые значения и функцию $itoa$, которая конвертирует число в строковое представление. Далее, используя вызовы, из функции $main$ нужно вызвать эти функции и посмотреть на результат. Исходный код функций приведен ниже.

```
itoa:
mov r8, 0
mov rax, rsi
mov ebx, 10
loop:
inc r8
mov edx, 0
div ebx
cmp rax, 0
jg loop
lea r9, [rdi + r8]
mov rax, rsi
loop2:
mov edx, 0
div ebx
add dl, '0'
dec r9
mov [r9], dl
cmp rax, 0
jg loop2
mov rax, r8
ret

print:
```

```
sub rsp, 56
mov rdx, rsi
mov rsi, rdi
mov rdi, 1
call write WRT ..plt
add rsp, 56
ret
```

Полный исходный код: https://github.com/ivanmakovetskiy/spo/blob/master/spo-main/syscall.asm

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гриценко, Ю. Б. Системное программное обеспечение : учебное пособие / Ю. Б. Гриценко. — Москва : ТУСУР, 2006. — 174 с.

2. Иванова, Н. Ю. Системное и прикладное программное обеспечение : учебное пособие / Н. Ю. Иванова, В. Г. Маняхина. — Москва : Прометей, 2011. — 202 с.

3. Адилов, Р. М. Системное программное обеспечение вычислительных систем : учебное пособие / Р. М. Адилов, Е. В. Грачёва, Н. Н. Короткова. — Пенза : ПензГТУ, 2012. — 118 с.

4. NASM. [Электронный ресурс]. URL: https://www.nasm.us/ (дата обращения: 11.04.2020).

5. Online x86 / x64 Assembler and Disassembler. [Электронный ресурс]. URL: https://defuse.ca/online-x86-assembler.htm (дата обращения: 10.03.2020).

6. Справочник по командам процессора x86. [Электронный ресурс]. URL: http://looch-disasm.narod.ru/refe01.htm (дата обращения: 10.03.2020).

7. COFF – OSDev Wiki. [Электронный ресурс]. URL: https://wiki.osdev.org/COFF (дата обращения: 10.03.2020).