Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 3

по дисциплине «Формальные грамматики и теории компиляторов»

Выполнил
студент гр. 4851003/80802 Сошнев М.Д.
<подпись>

Преподаватель Мясников А.В.
<подпись>

Санкт-Петербург
2021

Оглавление

1 ЦЕЛЬ РАБОТЫ

Понять принцип запуска и работы бинарного файла в системе.

2 ЗАДАЧА

Написать регистровую виртуальную машину, исполняющую бинарный файл, созданный при помощи программы из Лабораторной работы №2.

3 ХОД РАБОТЫ

Работу виртуальной машины можно разделить в 2 этапа — парсинг псевдоассемблерного кода и непосредственно его исполнение. Интуитивно хочется объединить эти этапы, чтобы программа работала по следующей схеме — считывала строчку из файла, выполняла, считывала следующую, выполняла и т. д. Но, так делать нельзя, потому что в таком случае непонятно, как работать с метками. Таким образом требуется сперва имитация «загрузки программы в память».

3.1 Парсинг псевдоассемблерного кода

Прочитанную строчку будем парсить с помощью регулярных выражений. Так как виртуальная машина была написана с использованием фреймворка QT, для регулярных выражений воспользуемся классом QRegExp. Прочитанная строчка может соответствовать 3 форматам: операция с 1 аргументом, операция с двумя аргументами или метка. Под каждый из этих форматов будем проверять по соответствующему регулярному выражению.

```
static QRegExp rxLabel(R"((\s*)(\w+)(\s*):(\s*))");
static QRegExp rxUnaryCommand(R"((\s*)(\w+)(\s+)(\w+)(\s*))");
static QRegExp rxBinaryCommand(R"((\s*)(\w+)(\s+)(\w+)(\s*),(\s*)(\w+)(\s*))");
```

Рисунок 1 — регулярные выражения для проверки строчки

В случае, если строчка не подходит ни под одно регулярное выражение, она будет напечатана на экран вместе с сообщением о неправильном синтаксисе и виртуальная машина будет завершена.

```
bool c = parseLine(line, cmd, argLeft, argRight, label);
if (!c) {
    qWarning() << "Wrong syntax:";
    qWarning() << line;

    return;
}
```

Рисунок 2 — поведение программы в случае неправильного синтаксиса

3.2 Хранение кода в памяти

Из прочитанной строчки будем формировать список команд — имитация загруженного в память кода. Для этого была сформирована следующая структура:

```
 8  struct Command {
 9      QString instruction;
10      QString argLeft;
11      QString argRight;
12
13      Command(const QString& instruction,
14              const QString& argLeft,
15              const QString& argRight)
16          : instruction(instruction)
17          , argLeft(argLeft)
18          , argRight(argRight) { }
19  };
20
21
22  class CommandList : public QList<Command> {
23  public:
24      CommandList();
25
```

Рисунок 3 — хранение псевдоассемблерного кода в памяти

При этом все метки из кода будут удалены, а в командах вида «jmp *метка*» будет заменена метка на номер ячейки массива CommandList, где лежит команда, на которую нужно «прыгнуть». После формирование данного массива, файл с псевдоассемблерным кодом можно закрыть.

3.3 Выполнение кода

За выполнение загруженного кода отвечает метод run класса CommandList. Имеется переменная EIP — имитация регистра (instruction pointer), счётчика инструкций. Метод run запустит последовательное выполнение команд — на каждой итерации будет запускаться команда commandList[EIP]. По умолчанию переменная EIP инкрементируется на каждой итерации, но, в случае, например команды JMP, в переменную EIP просто положится аргумент. Завершение выполнения кода может произойти в двух случаях — либо EIP указывает за пределы кода, либо встретилась команда «RET x».

3.4 Хранение данных

Во время выполнения кода данные могут хранится двумя способами — регистры и куча. В качестве имитации процессорных регистров заведём обычный массив.

```
37
38      int registers[32];
39

inline void setRegister(int regNum, int newValue) {
    registers[regNum - 1] = newValue;
}

inline int getRegister(int regNum) {
    return registers[regNum - 1];
}
```

Рисунок 4 — имитация хранения в регистрах

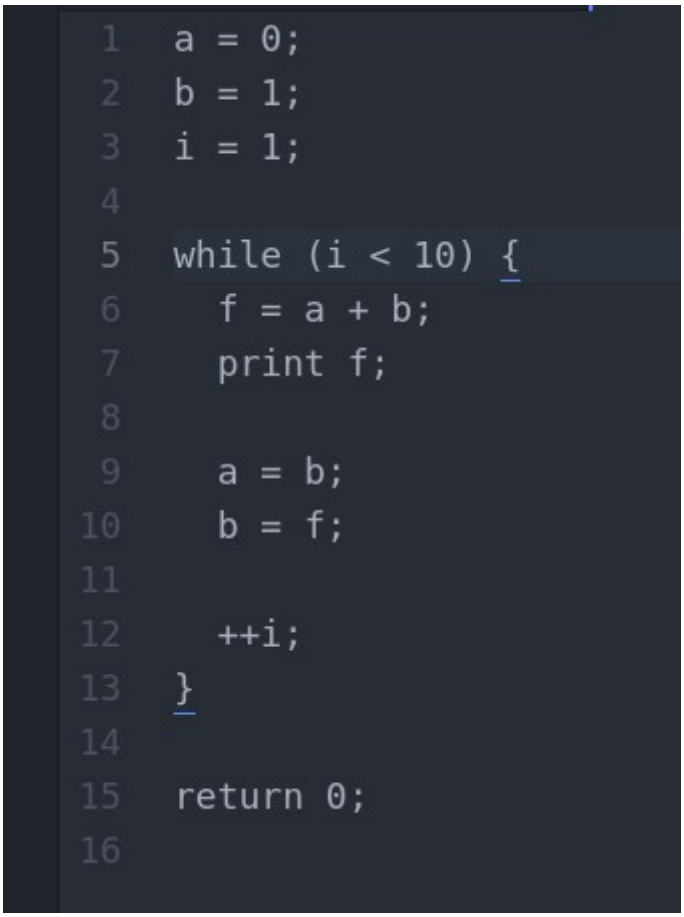
Для имитации кучи заведём «мапу» (класс Qmap, под капотом красно-чёрное дерево). Куча представляет собой множество пар: название переменной -значение переменой. Таким образом название переменной будет ключом мапы, а значение переменной — значением мапы.

```
// key = var name; value = var value
QMap<QString, int> varsMap;
```

Рисунок 5 — имитация кучи

4 ТЕСТИРОВАНИЕ

Напишем тестовую программу для виртуальной машины, пусть я хочу, чтоб программа вывела первые 10 чисел Фиббоначи.



```
1  a = 0;
2  b = 1;
3  i = 1;
4
5  while (i < 10) {
6    f = a + b;
7    print f;
8
9    a = b;
10   b = f;
11
12   ++i;
13 }
14
15 return 0;
16
```

Рисунок 6 — тестовая программа

Используя компилятор, написанный в лабораторной работе 2 транслируем эту программу в псевдоассемблерный код:

```
[maxim@maxim-81ag Work]$ ./compile tests/code.z -o tests/out.asm
OK
[maxim@maxim-81ag Work]$
```

```
1       MOV   a, 0
2       MOV   b, 1
3       MOV   i, 1
4
5   cycle_1_in:
6       MOV   R1, i
7       CMP   R1, 10
8       JGE   cycle_1_out
9       MOV   R1, a
10      ADD   R1, b
11      MOV   R2, R1
12      MOV   f, R2
13      MOV   R1, f
14      CALL  print
15      MOV   a, b
16      MOV   b, f
17      MOV   R1, i
18      INC   R1
19      MOV   i, R1
20      MOV   R1, i
21      JMP   cycle_1_in
22
23  cycle_1_out:
24      LDI   R1, 0
25      RET   R1
26
```

Рисунок 7, 8 — компиляция в псевдоассемблерный код

Теперь, соберём виртуальную машину из исходных текстов и запустим на ней полученный на предыдущем шаге код:

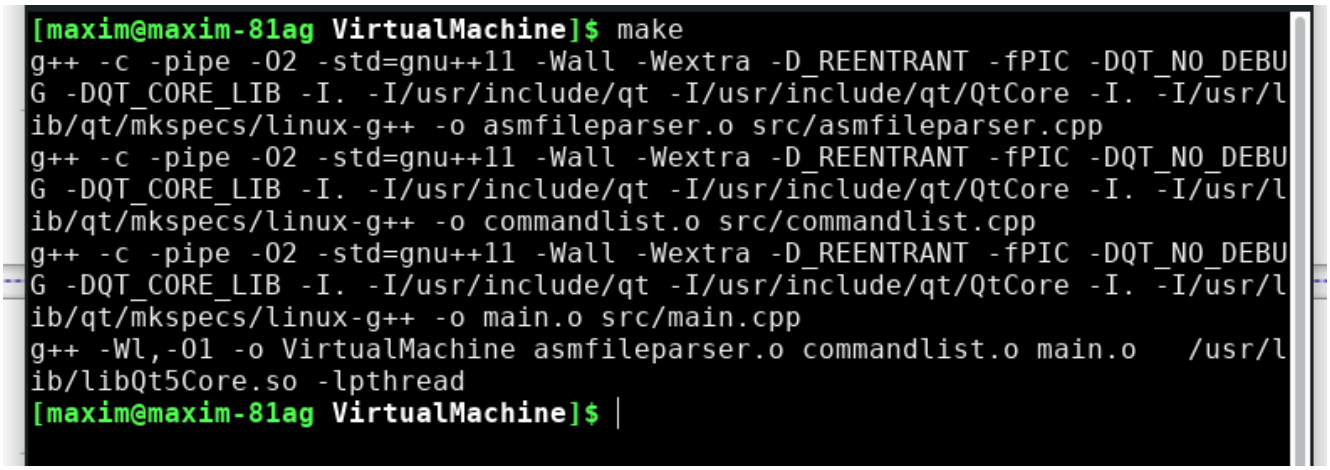


Рисунок 9 — успешная сборка виртуальной машины

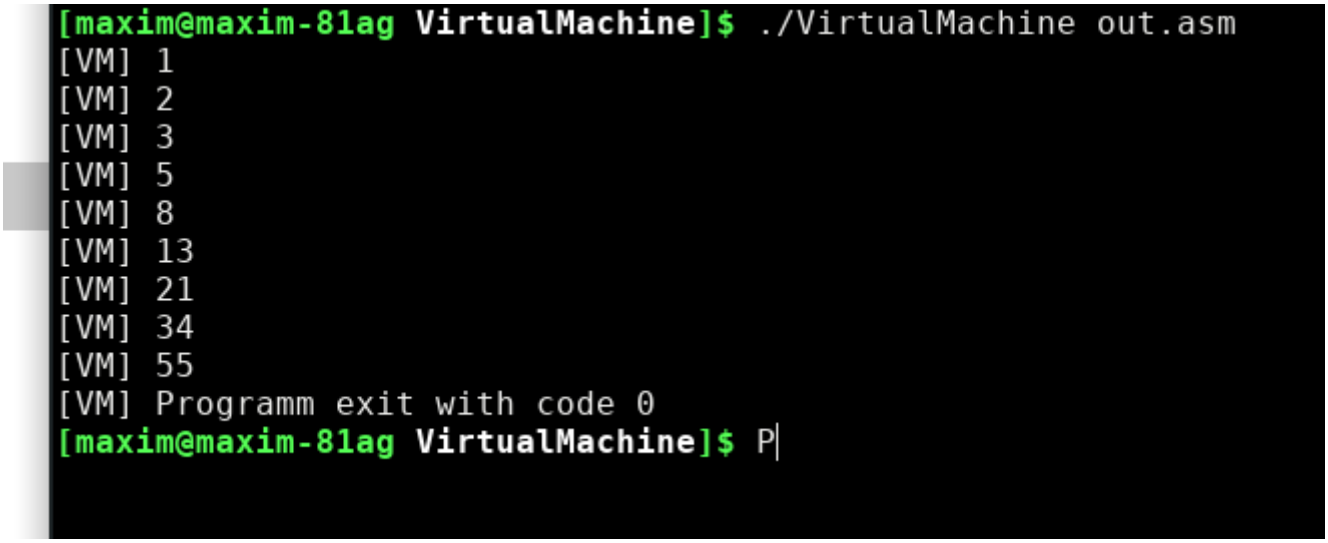


Рисунок 10 — исполнение кода на виртуальной машине

5 ВЫВОД

В ходе работы была написана виртуальная машина для исполнения псевдоассемблерного кода из прошлой лабораторной работы.

6 ПРИЛОЖЕНИЕ

Исходные тексты виртуальной машины

```
#include <QString>
#include <QFile>

#include <QDebug>

#include "asmfileparser.h"

int main(int argc, char** argv)
{
    if (2 != argc) {
        qWarning() << "Incorrect arguments";
        qWarning() << "Usage: ./VMachine [path asm]";

        return -1;
    }

    QString pathAsm = argv[1];
    QFile fileAsm(pathAsm);

    fileAsm.open(QIODevice::ReadOnly);
    if (!fileAsm.exists()) {
        qWarning() << "No such file: " << pathAsm;

        return -1;
    }

    AsmFileParser parser(fileAsm);
    parser.parseFile();

    fileAsm.close();

    CommandList commandList = parser.getCommandList();
    commandList.run();

    return 0;
}
```

```
#include "commandlist.h"
#include <QDebug>

CommandList::CommandList()
```

```
    : exit(false) { }

void CommandList::run() {
    EIP = 0;
    int lastInstructionNum = this->size();

    while (EIP < lastInstructionNum || !exit) {
        try {
            runCommand((*this)[EIP]);
        }  catch (QString& e) {
            qWarning() << e;
            return;
        }

        ++EIP;
    }

}

void CommandList::runCommand(Command& command) {
//    qDebug() << command.instruction << command.argLeft << command.argRight;

    bool c = jmpHandle(command);
    if (c)
        return;

    if (command.instruction == "MOV" || command.instruction == "LDI") {
        if (isRegister(command.argLeft)) {
            setRegister(command.argLeft.mid(1).toInt(),
                        getValue(command.argRight));
        }
        else {
            // this is var
            if (varsMap.contains(command.argLeft)) {
                // var is defined
                varsMap[command.argLeft] = getValue(command.argRight);
            } else {
                // this is var definition
                // insert new element to map
                varsMap.insert(command.argLeft,
                               getValue(command.argRight));
            }
```

```
    }
}

else if (command.instruction == "CMP") {
    flag = getValue(command.argLeft) - getValue(command.argRight);
    return;
}

else if (command.instruction == "CALL") {
    if (command.argLeft == "print") {
        printf("[VM] %d\n", getRegister(1));
    }
}

else if (command.instruction == "ADD") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue + getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "SUB") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue - getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "MUL") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue * getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "DIV") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue / getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "AND") {
```

```
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue & getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "OR") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue | getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "XOR") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue ^ getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "LS") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue << getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "RS") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue >> getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "MOD") {
    int regNum = command.argLeft.midRef(1).toInt();
    int oldValue = getRegister(regNum);
    int newValue = oldValue % getValue(command.argRight);
    setRegister(regNum, newValue);
}

else if (command.instruction == "INC") {
    int regNum = command.argLeft.midRef(1).toInt();
```

```
        int oldValue = getRegister(regNum);
        int newValue = ++oldValue;
        setRegister(regNum, newValue);
    }

    else if (command.instruction == "DEC") {
        int regNum = command.argLeft.midRef(1).toInt();
        int oldValue = getRegister(regNum);
        int newValue = --oldValue;
        setRegister(regNum, newValue);
    }

    else if (command.instruction == "RET") {
        int exitCode = getValue(command.argLeft);
        printf("[VM] Programm exit with code %d\n", exitCode);
        exit = true;
    }

    else {
        throw QString("Unknown command: " + command.instruction);
    }
}

bool CommandList::jmpHandle(Command& command) {
    if (command.instruction == "JMP") {
        EIP = getValue(command.argLeft) - 1;
        return true;
    }

    if (command.instruction == "JE") {
        if (flag == 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }

    if (command.instruction == "JNE") {
        if (flag != 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }
```

```
    if (command.instruction == "JG") {
        if (flag > 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }

    if (command.instruction == "JGE") {
        if (flag >= 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }

    if (command.instruction == "JL") {
        if (flag < 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }

    if (command.instruction == "JLE") {
        if (flag <= 0) {
            EIP = getValue(command.argLeft) - 1;
        }
        return true;
    }

    return false;
}

int CommandList::getValue(QString& from) {
    if (isRegister(from)) {
        return getRegister(from.mid(1).toInt());
    }

    if (isNum(from)) {
        return from.toInt();
    }

    // this is var

    if (!varsMap.contains(from)) {
```

```
        throw QString("Var " + from + "wasn't define");
    }

    return varsMap.value(from);
}

       #include <QRegExp>
#include <QDebug>

#include "asmfileparser.h"

AsmFileParser::AsmFileParser(QFile& file)
    : fileAsm(file) { }

void AsmFileParser::parseFile() {
    QString line, cmd, argLeft, argRight, label;

    while (!fileAsm.atEnd()) {
        // read all lines from file

        line = fileAsm.readLine();
        line = removeEndl(line);
        if (line.isEmpty()) {
            continue;
        }

        bool c = parseLine(line, cmd, argLeft, argRight, label);
        if (!c) {
            qWarning() << "Wrong syntax:";
            qWarning() << line;

            return;
        }

        if (!label.isEmpty()) {
            label = label.replace(":", "");
            labelMap.insert(label, commandList.size());

        } else {
            commandList.append(Command(cmd, argLeft, argRight));
        }
    }

    /* now in commands like [JMP label]
```

```
        we shoul change [lable] to number instruction
        (take it from labelMap */

    int i = 1;
    for (Command& command : commandList) {
        if (command.instruction == "JMP"
            || command.instruction == "JE"
            || command.instruction == "JNE"
            || command.instruction == "JG"
            || command.instruction == "JL"
            || command.instruction == "JGE"
            || command.instruction == "JLE") {

            command.argLeft = QString::number(labelMap.value(command.argLeft));
        }

        ++i;
    }
}

bool AsmFileParser::parseLine(
        QString& src,
        QString& cmd,
        QString& argLeft,
        QString& argRight,
        QString& label) {
    static QRegExp rxLabel(R"((\s*)(\w+)(\s*):(\s*))");
    static QRegExp rxUnaryCommand(R"((\s*)(\w+)(\s+)(\w+)(\s*))");
        static  QRegExp  rxBinaryCommand(R"((\s*)(\w+)(\s+)(\w+)(\s*),(\s*)(\w+)(\
s*))");

    cmd.clear();
    argLeft.clear();
    argRight.clear();
    label.clear();

    src = src.replace("\t", " ");

    if ((-1 == rxUnaryCommand.indexIn(src))
        && (-1 == rxLabel.indexIn(src))) {
        return false;
    }
```

```
    if ((rxUnaryCommand.matchedLength() != src.size())
            && (-1 == rxBinaryCommand.indexIn(src))
            && (-1 == rxLabel.indexIn(src))) {
        return false;
    }

    src = src.replace(" ", ",");
    QStringList list = src.split(",");
    list.removeAll(QString(""));

    if (list.size() == 1) {
        label = *list.begin();
        return true;
    }

    cmd = list[0];
    argLeft = list[1];

    if (list.size() > 2) {
        argRight = list[2];
    }

    return true;
}
```

```
       #ifndef COMMANDLIST_H
#define COMMANDLIST_H

#include <QString>
#include <QList>
#include <QMap>

struct Command {
    QString instruction;
    QString argLeft;
    QString argRight;

    Command(const QString& instruction,
            const QString& argLeft,
            const QString& argRight)
        : instruction(instruction)
        , argLeft(argLeft)
        , argRight(argRight) { }
```

```
};

class CommandList : public QList<Command> {
public:
    CommandList();

    void run();

private:
    int EIP;
    int flag;   // for commands like JNE

    bool exit;

    // key = var name; value = var value
    QMap<QString, int> varsMap;

    int registers[32];

private:
    void runCommand(Command& command);

    bool jmpHandle(Command& command);

    int getValue(QString& from);

    inline bool isRegister(QString& s) {
        static QRegExp rxRegister(R"(R(\d+))");
        return rxRegister.indexIn(s) != -1;
    }

    inline bool isNum(QString& s) {
        static QRegExp rxNum(R"(\d+)");
        return rxNum.indexIn(s) != -1;
    }

    inline void setRegister(int regNum, int newValue) {
        registers[regNum - 1] = newValue;
    }

    inline int getRegister(int regNum) {
        return registers[regNum - 1];
    }
};
```

```
#endif // COMMANDLIST_H

       #ifndef VIRTUAL_MACHINE_ASMCOMMANDHANDLER_H
#define VIRTUAL_MACHINE_ASMCOMMANDHANDLER_H

#include <QFile>
#include <QMap>

#include "commandlist.h"

class AsmFileParser {
public:
    explicit AsmFileParser(QFile& file);

    void parseFile();

    inline CommandList getCommandList() {
        return std::move(commandList);
    }

private:
    bool parseLine(QString& src,
                   QString& cmd,
                   QString& argLeft,
                   QString& argRight,
                   QString& label);

    inline QString removeEndl(QString& line) {
        return line.replace("\r", "").replace("\n", "");
    }

private:
    QFile& fileAsm;
    CommandList commandList;

    // key = label name; value = instruction number
    QMap<QString, int> labelMap;
};

#endif //VIRTUAL_MACHINE_ASMCOMMANDHANDLER_H
```