DV1655/6 - Assignment 3

Intermediate representation, code generation, and interpretation

Suejb Memeti
Blekinge Institute of Technology

February 25, 2022

1 Introduction

In this assignment you are going to :

1. Traverse the AST and generate the Intermediate Representation - IR (described in Section 2),
2. Traverse the IR and generate the target code (described in Section 3), and
3. Interpret the target code (described in Section 4).

Accordingly, this assignment is split into three parts, (1) Intermediate Representation, (2) Code Generation, and (3) Interpretation.

The goal of the first part is to traverse the AST to construct the intermediate representation (IR), which is a data structure that contains basic blocks and three-address code (TAC) instructions.

The goal of the second part of this assignment is to traverse the IR and for each of the three-address code instructions in the basic building blocks generate java byte-code.

The goal of the third part of is to write a stack-based machine interpreter, which can read and interpret java byte-code instructions.

The examination of this project is done through demonstrations during the lab sessions. It is expected that you submit the source code in Canvas. The source code should be compressed in a zip/tar file.

The deadline for demonstrating this assignment is the last lab session, i.e., March 24, 2022.

The solution should be implemented using C or C++, and it should be compiled and executed correctly on a Unix-based operating system.

1.1 Laboratory groups

You are encouraged to work in groups of two students. Groups larger than two are not accepted. Groups of one student are accepted, but not encouraged. The project is designed for groups of two students.

Discussion and help between laboratory groups are encouraged. However, you should be careful to not cross the border to plagiarism, see section 1.3 below.

1.2 Lecture support

In Canvas (the course management page), you should be able to find the Intermediate Representation lecture, which is related to the first part of the assignment. In that lecture we introduce the general concepts and theory for intermediate representation. We briefly describe the three address instructions and basic blocks intermediate representation and provide examples of how we can traverse the AST and construct the IR.

The code generation and interpretation lecture corresponds to the second and third part of this assignment. In that lecture, we describe translation process from a control flow graph (in terms of basic blocks and three-address instructions) into byte-code. Furthermore, we describe the stack machine-based execution of the byte-code instruction (i.e. interpretation).

1.3 Plagiarism

All work that is not your own should be properly referenced. If not, it will be considered as cheating and reported as such to the university disciplinary board.

2 Part 1: Intermediate Representation (IR)

The first step of this assignment is to traverse the abstract syntax tree and generate the corresponding low-level intermediate representation.

At this point, we assume that the provided source code is lexically, syntactically,and semantically correct. To do so, check that the input test classes that you use in this stage are compiled by Java.

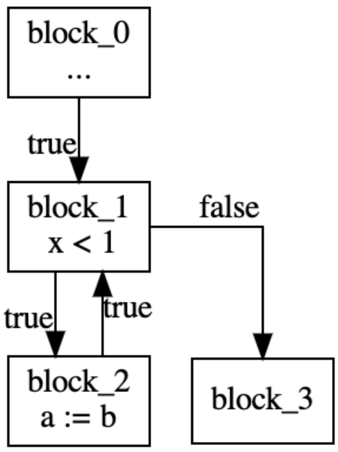
Design/Implement data structures (class hierarchy) similar to what we proposed in the Intermediate Representation lecture to store the three address instructions and the basic blocks (See slide: 11).

Write a tree traversal for generating three address instructions for expressions, and basic blocks for statements. Note that we distinguish two types of language constructs for which we need to perform some actions when generating the IR,expressions and statements. As mentioned in the lecture, expressions do not generate new basic blocks in the control flow graph, instead they simply add three address instructions to the current block, whereas statements do generate new blocks most of the times.

Write a function that prints the basic blocks into a dot format, similarly to what we have seen in the slides, and similarly to what we have done when generating the graphviz output for the AST. A simple example of what the output of the dot file should look like is provided in the example below.

Table 1: An example of a dot file (on the left) and the generated output (on the right) for a simple CFG. It corresponds to the translation of the while statements example from slide 20 of the Intermediate Representation lecture.

```
digraph {
    graph [splines=ortho]
    node  [shape=box];
    block_0 [label="block_0\n...\n"];
    block_0 -> block_1 [xlabel="true"];
    block_1 [ shape=box label="block_1\n␣x␣<␣1\n"
        ];
    block_1 -> block_2 [xlabel="true"];
    block_1 -> block_3 [xlabel="false"];
    block_2 [ shape=box label="block_2\na␣:=␣b\n"
        ];
    block_2 -> block_1 [xlabel="true"];
    block_3 [ shape=box label="block_3\n"];
}
```



2.0.1 Recommended approach

The recommended approach for constructing the intermediate representation is as follows:

- Create the objects/classes to store the information for the three-address code and basic blocks
- Write a method that prints the CFG in a dot file.
- Start translating expressions. The result should be one basic block with a list of three address code instructions.
- Continue with the translation of statements. The result should be multiple basic blocks linked together, each basic block may have multiple three address code instructions.

3 Part 2: Target code generation - Java byte-code

The target code that we will use in this course will be a simplified version and similar to what the Java byte-code looks like. Note that, the generated target code is not executable by a real Java Virtual Machine (JVM). The aim is to simulate the way how byte-code is generated by the Java compiler and then how such code is interpreted by JVM. For students that are interested to see what are the differences between our version of the byte-code and the actual Java byte-code, Section A in the Appendix, shows an example and summarizes the main differences.

In the second assignment, we simplified the MiniJava grammar by disregarding inheritance and polymorphism. In this assignment, we further simplify the MiniJava grammar in the following ways:

- We limit the type of variables to *integers* and *Booleans*, that is, we disregard *arrays* and *Identifier* types (i.e., class objects). We treat *Booleans* as *integers* (*true = 1*, and *false = 0*).
- We disregard public variables in a class declaration, that is, a class declaration will have only methods.

Such limitations apply only to step 2 and 3. You still need to be able to generate the correct IR for all MiniJava programs.

Assuming that you have a list of entry points in your IR, where each entry point corresponds to the first block of a method, to generate byte-code you need to do the following:

- Loop over the list of entry points, and traverse the CFG starting from the entry point
- Visit all TAC instructions of the current block, and generate byte-code instructions (see Table 1 for the full list of instructions needed)
- Visit the *trueExit* block and the *falseExit* block, if not null and not already visited.
- Keep track of the visited blocks to avoid generating target code for already visited blocks (example, in the while loop statement, the true exit of the loop body block points to the loop header, the loop body and the loop header should be visited only once during code generation)

The main byte-code instructions that will be needed for this assignment include (1) instructions to load and store values into the stack *(iload, iconst, istore*); (2) instructions to perform arithmetic and logic operations (*iadd, isub, imul, idiv, ilt, igt, ieq, iand,ior, inot*); (3) jump instructions (*goto, iffalse goto*); (4) method calls (*invokevirtual*); and(5) *return, print*, and *exit* instructions. The full set of instructions and their description is listed in Table 1.

The output of this step is a file that contains either:

- A readable java byte-code, similar to the examples in the table above
- A serialized file that contains the necessary information to reconstruct the object files that represent the byte-code instructions. Note that you still need a method for pretty printing the byte-code.

Note that the generated class files does not necessarily need to be compatible with the examples that I have provided. It should not necessarily be the same with the format that other groups will use either. It is important though that the format of the class file is readable and executable by the interpreter (see Part 3).

3.0.1 Recommended approach

For this part of the assignment, I suggest the following approach:

- You may need to design data structures for storing the byte-code instructions. Add a function that can write such instruction to a file or simply serialize an object to a file.
- Start with the simplest example (E.java), which has two simple print statements, each with a specific expression, an arithmetic expression and a logical expression.
- Continue adding more features, such as different statements, including assignment statements, if-else statements and while statements (D1.java and D2.java).

Instruction	Description
iload n	Push integer value stored in local variable n.
iconst v	Push integer value v.
istore n	Pop value v and store it in local variable n.
iadd	Pop value v1 and v2. Push v2 + v1.
isub	Pop value v1 and v2. Push v2 - v1.
imul	Pop value v1 and v2. Push v2 * v1.
idiv	Pop value v1 and v2. Push v2 / v1.
ilt	Pop value v1 and v2. Push 1 if v2 <v1, else push 0.
igt	Pop value v1 and v2. Push 1 if v2 >v1, else push 0.
ieq	Pop value v1 and v2. Push 1 if v2 == v1, else push 0.
iand	Pop value v1 and v2. Push 0 if v1 * v2 == 0, else push 1.
ior	Pop value v1 and v2. Push 0 if v1 + v2 == 0, else push 1.
inot	Pop value v. Push 1 if v == 0, else push 0.
goto i	Jump to instruction labeled i unconditionally.
iffalse goto i	Pop value v from the data stack. If v == 0 jump to instruction labeled i, else continue with the following instruction.
invokevirtual m	Push current activation to the activation stack and switch to the method having qualified name m.
ireturn	Pop the activation from the activation stack and continue.
print	Pop the value from the data stack and print it.
stop	Execution completed.

- Add the functionality to generate code for recursive calls (D3.java).
- Add the functionality to generate code for nested if-else and while statements (C1.java and C2.java).
- Add functionality for handling multiple methods and classes (B.java).
- Add functionality for generating code for multi-level nested method calls (A.java).

4 Part 3: Stack-machine based interpretation

The interpreter is a ***separate*** program, which reads the output generated from step 2,and executes the instructions inside the program.

Depending on what the output file from step 2 is, you may need to:

- Read the file line by line and reconstruct the byte-code instructions and execute them, or
- Read the file, deserialize it, and execute the instructions.

Please refer to the descriptions in Table 1 to get an idea of how each instruction should be interpreted.

I strongly recommend listening to the code generation and interpretation lecture to get a better understanding of how stack-machine based execution works.

Note that the stack-machine based interpreter uses two stacks, the data stack that is used to keep track of the data values, and the activation stack that keeps track of the previously called but not completed methods. This means that a recursive call results with multiple items in the activation stack. For example, on *invokevirtual* instructions we push to the activation stack, on *return* instructions we pop from the activation stack.

4.0.1 Recommended approach

The recommended approach for the stack-machine based interpreter is similar to the ones related to code generation, which basically mean that you start from the simplest examples and build toward the more complex language constructs.

- You may need to write a function that reads a file and constructs the byte-code instructions from it, or simply deserialize an object.
- Start with interpreting the simplest language constructs, i.e. expressions, including arithmetic and logical expressions (E.java).
- Continue adding more features to the interpreter, such as interpreting different statements, including assignment statements, if-else statements and while statements (D1.java and D2.java).
- Add the functionality to interpret instructions related to recursive calls (D3.java).
- Add the functionality to interpret instructions related to nested if-else and while statements (C1.java and C2.java).
- Add functionality for interpret byte-code blocks that correspond to multiple methods and classes (B.java).
- Add functionality for interpreting instructions for multi-level nested method calls (A.java).

5 Testing

For the first part of the assignment, that is constructing the intermediate representation, you can use the test classes from the Cambridge website. Constructing the IR for any given MiniJava program is part of the minimum requirements to pass this assignment. For the second and third part of this assignment, I provide a set of examples that can be used for testing your approach. You are encouraged to write other test classes to test various functionalities of your compiler.

6 Examination

During the demonstration:

- Your compiler should be easily compiled using the Makefile.

- You should be able to explain specific parts of the solution. For example, part sof the code that are responsible for traversing the AST and generating the IR. I may ask to explain how the IR is built for a particular feature of the MiniJava. You may also expect questions related to how the code generator and the interpreter works.
- You are expected to show the IR visually for each of the test classes used (from the Cambridge website).
- You are expected to show the generated byte-code, and eventually explain parts of it.
- You are expected to run each of the test programs and show that the result of the interpretation of the generated code is the same as the one of the java compiler.
  - Compile each test program with java, and observe the result
  - Compile each test program with your compiler, and observe the result

7 Grading Scheme

The minimum requirements for this assignment include:

- The IR should be generated for all of the MiniJava programs (i.e. all test classes from the Cambridge website) and pretty printed in .dot file.
- The code generator should work for the E.java example.
- The interpreter should be able to interpret the generated code for the E.java example.

Higher grades can be achieved by implementing the code generator and interpreter for the rest of the MiniJava features available in the other java class examples.

A MiniJava byte-code vs Java byte-code

An example that highlights the differences between our version of byte-code and the java byte-code is shown below, in Table 2. Here we use a simple example, that calculates the sum of all number from 0 to N (in this case N = 100). Note that here we present two alternatives for our byte-code.

In the first one we use labels, such as Sum.main or 0, 1, 2, ...; Lettered labels correspond to the qualified names of methods, whereas numbered labels inside a method correspond to the instruction number.

In most cases the labeled numbers will not be used. The only ones that have meaning are the ones that are used inside a goto instruction (in this case in *Test.Sum*, *label 3* and *16*).

Alternatively, you may use the second representation, where we only add labels for method names and blocks that are used in goto instructions (in this case, in *Test.Sum*, *block_2* and *block_3*).

Another difference between representation 1 and 2, is that in the first one we use indices to access local variables (example, #0 corresponds to num; and #1 to sum),whereas in the second representation, we use the variable names (example num, sum). Note that variable names are unique inside a scope. You could also combine using numbered labels for instructions and variable names. All variants area cceptable.

Table 2: An example of a dot file (on the left) and the generated output (on the right) for a simple CFG. It corresponds to the translation of the while statements example from slide 20 of the Intermediate Representation lecture.

Test example

```
class Sum {
 public static void
    main(String[] a)
    {
  System.out.println
     (new Test().Sum
     (100));
 }
}
class Test {
 public int Sum(int
    num) {
  int sum;
  sum = 0;
  while (0 < num) {
   sum = sum + num;
   num = num - 1;
  }
  return sum;
 }
}
```

Our byte-code (1)

```
Sum.main
0 iconst 100
1 invokevirtual Test
   .Sum
2 print
3 stop

Test.Sum
0 istore #0
1 iconst 0
2 istore #1
3 iconst 0
4 iload #0
5 ilt
6 if_false 16
7 iload #1
8 iload #0
9 iadd
10 istore #1
11 iload #0
12 iconst 1
13 isub
14 istore #0
15 goto 3
16 iload #1
17 ireturn
```

Our byte-code (2)

```
Sum.main:
 iconst 100
 invokevirtual Test.
    Sum
 print
 stop

Test.Sum:
 istore num
 iconst 0
 istore sum
 block_2:
  iconst 0
  iload num
  ilt
  iffalse goto
     block_3
  iload sum
  iload num
  iadd
  istore sum
  iload num
  iconst 1
  isub
  istore num
  goto block_2
 block_3:
  iload sum
  ireturn
```

Java Byte-code

```
public class Sum {
 ... //constructor

 public static void
    main(...);
 0 getstatic java.
    lang.System.out:
     ... [16]
 3 new Test [18]
 6 dup
 7 invokespecial
    Test() [19]
 10 bipush 100
 12 invokevirtual
    Test.Sum(int):
    int [22]
 15 invokevirtual
 java.io.PrintStream
    .println(int) :
    void [28]
 18 return
}
```

```
public class Test {
 ... //constructor

 public int Sum(int
    arg0);
 0 iconst_0
 1 istore_2
 2 iconst_0
 3 iload_1 [arg0]
 4 if_icmplt 11
 7 iconst_0
 8 goto 12
 11 iconst_1
 12 ifeq 26
 15 iload_2
 16 iload_1 [arg0]
 17 iadd
 18 istore_2
 19 iload_1 [arg0]
 20 iconst_1
 21 isub
 22 istore_1 [arg0]
 23 goto 2
 26 iload_2
 27 ireturn
 ...
}
```