COM S 440/540 Project part 4

Code generation: expressions

1 Requirements for part 4

When executed with a mode of 4, your compiler should read the specified input file, and check it for correctness (including type checking) as done in part 3. If there are no errors, then your compiler should output an equivalent program in our target language (see Section 2). For this part of the project, your compiler must generate correct code for only some types of statements: namely, expressions (except for the ternary operator), including function calls and returns. Control flow (conditionals, the ternary operator, and loops) will be required for the next and final part of the project. As usual, any error messages should be written to standard error, and your compiler may make a "best effort" to continue processing the input file, or exit. Errors detected in this part of the project should have format

> `Code generation error in file` *filename* `line` *line number*
> *Description*

where line numbers will be specified below if not obvious. If there are no errors, then your compiler should build an output file with name *input*`.j`, when run on a source file named *input*`.c`.

2 The target language

The target language is Java assembly language (see for example `https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions`), as read by the Krakatau Java assembler (written in Python and available on github at `https://github.com/Storyyeller/Krakatau`). Generated assembly code should be for a class whose name matches the source file being compiled, with global variables becoming static members of the class, and functions becoming static methods of the class. The produced class should be derived from `java/lang/Object`.

The output file to be read by the assembler is plain ASCII text. Comments, which begin with a semicolon and end with a newline character, may appear on an empty line or after a valid line (with space in between the line and the comment). Students are encouraged to add blank lines, formatting, and comment lines as appropriate to help with debugging efforts.

Your compiler **may not** invoke the Java compiler. However, it is perfectly fine to "reverse engineer" the Java compiler, to help you figure out which assembly instructions to use for translation. For instance, you could write Java code, compile it, then use the Krakatau disassembler to view the assembly code generated by the Java compiler, to help you learn and understand the Java bytecode instructions.

2.1 Class information

At minimum, you need `.class` and `.super` lines at the beginning of output. For source file *input.c*, these lines should be

```
.class public input
.super java/lang/Object
```

2.2 Global variables

Global variables may appear in output in any order (recommended is to output them as encountered) as long as they appear before they are used. Again, these should be public, static members. These have format

```
.field public static varname T
```

where *varname* is the global variable name, and T is the type information.

2.3 User-defined functions

Each function with a body in the C source should cause the generation of (Java assembly for) a static method. User-defined methods should appear before the special ones (see below). It is recommended, but not required, to generate assembly for functions in the same order as they appear in the input file, and as they are encountered.

2.4 Java's main method

Your compiler should produce (Java assembly for) a Java `main()` method. In Java source, the method would have prototype

```
public static void main(String args[])
```

and its job is to invoke the C source's static `main()` method, with prototype

```
int main()
```

(you do not need to handle the other legal prototypes for main in C). The value returned by C's `main()` should be passed back to the system; this should be done by invoking Java's `System.exit()` method, passing the return code as a parameter.

2.5 Special method `<init>`

Constructors in Java assembly have the special name `<init>`, and a return type of `void`. Your compiler should generate an `<init>` method for an empty constructor (i.e., no parameters). Since everything will be static, the constructor does not need to do anything except invoke the constructor for base class `java/lang/Object`.

2.6 Special method `<clinit>`

The special method `<clinit>` is used to initialize any static members that require initialization. For the C source code, this includes all global variables that were initialized when declared (this was extra credit in parts 2 and 3), and all global arrays. If there are no such items to initialize, then method `<clinit>` may be omitted.

3 I/O

To assist students (and instructors) in debugging and testing compilers, the following functions should be "hard coded" into the symbol table as the ComS 440 standard library.

- `int getchar()`: This should behave the same as `getchar()` in `stdio.h`.
- `int putchar(int c)`: This should behave the same as `putchar()` in `stdio.h`.
- `int getint()`: Read an integer from standard input, and return it.
- `void putint(int x)`: Write an integer to standard output.
- `float getfloat()`: Read a real value from standard input, and return it as a float.

- `void putfloat(float x)` Write a float value to standard output.
- `void putstring(const char s[])`: Write a null-terminated array of characters to standard output.

The Java source for these (public, static) methods will be provided, as `lib440.java`. If you compile this to `lib440.class`, then any `.j` files produced by your (working) compiler may be assembled and then run in Java. The `lib440` class also contains method

```
public static char[] java2c(String s)
```

for converting a Java string into a null-terminated array of characters. Your compiler may generate Java assembly calls to this method as needed, in particular to convert from a string literal to a character array. See the example code shown in Section 5.

Additionally, the C source for these functions will be provided as `lib440.c`. Any test files (instructor-provided or ones you write yourself) may be tested with a production C compiler (e.g., gcc or clang), by concatenating `lib440.c` with the test file before compiling.

4 Checking your generated code

A number of test scripts are provided; these are used to assist with grading, so students are encouraged to ensure that their compilers work well with the scripts. As with previous parts of the project, scripts typically require the command to execute your compiler as its first argument (enclosed in double quotes), followed by any source files to test.

4.1 Test by running

The script `RunTest.sh` will test your compiler on input source files by assembling the file produced by your compiler (using the Krakatau assembler) into a `.class` file, and then running the `.class` file on a JVM. It then compares the output(s) of running your `.class` file against expected outputs, which are obtained using the `gcc` compiler.

You are enouraged to generate your own test programs and inputs, using a C source file `base.c`, and inputs `base.input01`, `base.input02`, etc. (not needed if `base.c` does not read any input). Use the `-G` switch to `RunTest.sh` to generate the `gcc` outputs.

4.2 Test class and method output

To test just the `.class`, `.super`, `.method`, `.code`, and `.end` portions of the compiler output, use script `DotTest.sh`. This compares against expected output, as generated by one of the Instructor compilers.

4.3 Test assembly against spec

To test the generated assembly instructions, use script `AsmTest.sh`. Because several different instruction sequences are often possible to correctly implement the C source, this script will compare your generated assembly against hand-crafted "specs", containing one or more tests. A test passes when the generated code matches a specified pattern.

5 Examples

5.1 Input: `hello.c`

```
1
2  int main()
3  {
4      putstring("Hello, world!\n");
```

```
5       return 2;
6       putstring("This is dead code\n");
7       return 0;
8   }
```

5.2 Basic output: `hello.j`

```
1   .class public hello
2   .super java/lang/Object
3
4
5   .method public static main : ()I
6       .code stack 1 locals 0
7           ; expression statement at INPUTS/hello.c line 4
8           ldc "Hello, world!\n"
9           invokestatic Method lib440 java2c (Ljava/lang/String;)[C
10          invokestatic Method lib440 putstring ([C)V
11          ; return statement at INPUTS/hello.c line 5
12          iconst_2
13          ireturn
14          ; expression statement at INPUTS/hello.c line 6
15          ldc "This is dead code\n"
16          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
17          invokestatic Method lib440 putstring ([C)V
18          ; return statement at INPUTS/hello.c line 7
19          iconst_0
20          ireturn
21      .end code
22  .end method
23
24
25  .method <init> : ()V
26      .code stack 1 locals 1
27          aload_0
28          invokespecial Method java/lang/Object <init> ()V
29          return
30      .end code
31  .end method
32
33  .method public static main : ([Ljava/lang/String;)V
34      .code stack 1 locals 1
35          invokestatic Method hello main ()I
36          invokestatic Method java/lang/System exit (I)V
37          return
38      .end code
39  .end method
```

5.3 Extra output: `hello.j`

This compiler implemented dead code removal, so instructions after the first `return` statement (lines 6 and 7) do not generate assembly instructions. (Actually, the compiler comments those out, which is easier to test, and has the same effect.)

```
1   .class public hello
2   .super java/lang/Object
```

```
3
4
5  .method public static main : ()I
6      .code stack 1 locals 0
7          ; expression statement at INPUTS/hello.c line 4
8          ldc "Hello, world!\n"
9          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
10         invokestatic Method lib440 putstring ([C)V
11         ; return at INPUTS/hello.c line 5
12         iconst_2
13         ireturn
14         ;DEAD   ; expression statement at INPUTS/hello.c line 6
15         ;DEAD   ldc "This is dead code\n"
16         ;DEAD   invokestatic Method lib440 java2c (Ljava/lang/String;)[C
17         ;DEAD   invokestatic Method lib440 putstring ([C)V
18         ;DEAD   ; return at INPUTS/hello.c line 7
19         ;DEAD   iconst_0
20         ;DEAD   ireturn
21         ;DEAD   ; implicit return at INPUTS/hello.c line 8
22         ;DEAD   return
23     .end code
24 .end method
25
26
27 .method <init> : ()V
28     .code stack 1 locals 1
29         aload_0
30         invokespecial Method java/lang/Object <init> ()V
31         return
32     .end code
33 .end method
34
35 .method public static main : ([Ljava/lang/String;)V
36     .code stack 1 locals 1
37         invokestatic Method hello main ()I
38         invokestatic Method java/lang/System exit (I)V
39         return
40     .end code
41 .end method
```

5.4 Input: `sum1.c`

```
1
2  int main()
3  {
4      int a, b;
5      putstring("Enter two integers\n");
6      a = getint();
7      b = getint();
8      putint(a);
9      putstring(" + ");
10     putint(b);
11     putstring(" = ");
```

```
12      putint(a+b);
13      putchar(10);
14      return 0;
15  }
```

5.5 Basic output: `sum1.j`

Technically, the assignment statements on lines 6 and 7 of `sum1.c` are expressions that should produce integers. This compiler leaves a copy of those integers on the stack (lines 13 and 18 of `sum1.j`), and then pops them off (lines 15 and 20).

```
1   .class public sum1
2   .super java/lang/Object
3
4
5   .method public static main : ()I
6       .code stack 2 locals 2
7           ; expression statement at INPUTS/sum1.c line 5
8           ldc "Enter two integers\n"
9           invokestatic Method lib440 java2c (Ljava/lang/String;)[C
10          invokestatic Method lib440 putstring ([C)V
11          ; expression statement at INPUTS/sum1.c line 6
12          invokestatic Method lib440 getint ()I
13          dup
14          istore_0 ; a
15          pop
16          ; expression statement at INPUTS/sum1.c line 7
17          invokestatic Method lib440 getint ()I
18          dup
19          istore_1 ; b
20          pop
21          ; expression statement at INPUTS/sum1.c line 8
22          iload_0 ; a
23          invokestatic Method lib440 putint (I)V
24          ; expression statement at INPUTS/sum1.c line 9
25          ldc " + "
26          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
27          invokestatic Method lib440 putstring ([C)V
28          ; expression statement at INPUTS/sum1.c line 10
29          iload_1 ; b
30          invokestatic Method lib440 putint (I)V
31          ; expression statement at INPUTS/sum1.c line 11
32          ldc " = "
33          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
34          invokestatic Method lib440 putstring ([C)V
35          ; expression statement at INPUTS/sum1.c line 12
36          iload_0 ; a
37          iload_1 ; b
38          iadd
39          invokestatic Method lib440 putint (I)V
40          ; expression statement at INPUTS/sum1.c line 13
41          bipush 10
42          invokestatic Method lib440 putchar (I)I
43          pop
44          ; return statement at INPUTS/sum1.c line 14
```

```
45          iconst_0
46          ireturn
47      .end code
48  .end method
49
50
51  .method <init> : ()V
52      .code stack 1 locals 1
53          aload_0
54          invokespecial Method java/lang/Object <init> ()V
55          return
56      .end code
57  .end method
58
59  .method public static main : ([Ljava/lang/String;)V
60      .code stack 1 locals 1
61          invokestatic Method sum1 main ()I
62          invokestatic Method java/lang/System exit (I)V
63          return
64      .end code
65  .end method
```

5.6 Extra output: `sum1.j`

This compiler implements smarter stack management, and realizes that the values for the assignment statements on lines 6 and 7 of `sum1.c` are not used. Therefore the values are not left on the stack, and there is no need to pop them off.

```
1   .class public sum1
2   .super java/lang/Object
3
4
5   .method public static main : ()I
6       .code stack 2 locals 2
7           ; expression statement at INPUTS/sum1.c line 5
8           ldc "Enter two integers\n"
9           invokestatic Method lib440 java2c (Ljava/lang/String;)[C
10          invokestatic Method lib440 putstring ([C)V
11          ; expression statement at INPUTS/sum1.c line 6
12          invokestatic Method lib440 getint ()I
13          istore_0 ; a
14          ; expression statement at INPUTS/sum1.c line 7
15          invokestatic Method lib440 getint ()I
16          istore_1 ; b
17          ; expression statement at INPUTS/sum1.c line 8
18          iload_0 ; a
19          invokestatic Method lib440 putint (I)V
20          ; expression statement at INPUTS/sum1.c line 9
21          ldc " + "
22          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
23          invokestatic Method lib440 putstring ([C)V
24          ; expression statement at INPUTS/sum1.c line 10
25          iload_1 ; b
26          invokestatic Method lib440 putint (I)V
```

```
27          ; expression statement at INPUTS/sum1.c line 11
28          ldc " = "
29          invokestatic Method lib440 java2c (Ljava/lang/String;)[C
30          invokestatic Method lib440 putstring ([C)V
31          ; expression statement at INPUTS/sum1.c line 12
32          iload_0 ; a
33          iload_1 ; b
34          iadd
35          invokestatic Method lib440 putint (I)V
36          ; expression statement at INPUTS/sum1.c line 13
37          bipush 10
38          invokestatic Method lib440 putchar (I)I
39          pop
40          ; return at INPUTS/sum1.c line 14
41          iconst_0
42          ireturn
43          ;DEAD   ; implicit return at INPUTS/sum1.c line 15
44          ;DEAD   return
45      .end code
46  .end method
47
48
49  .method <init> : ()V
50      .code stack 1 locals 1
51          aload_0
52          invokespecial Method java/lang/Object <init> ()V
53          return
54      .end code
55  .end method
56
57  .method public static main : ([Ljava/lang/String;)V
58      .code stack 1 locals 1
59          invokestatic Method sum1 main ()I
60          invokestatic Method java/lang/System exit (I)V
61          return
62      .end code
63  .end method
```

6 Grading

For all students: implement as many or as few features listed below as you wish, but keep in mind that some features will make testing your code *much* easier (features needed to test your code for part 5 are marked with †), and a deficit of points will impact your overall grade. Excess points will count as extra credit.

Points	Description
15	**Documentation**
3	`README.txt` How to build the compiler and documentation. Updated to show which part 4 features are implemented.
12	`developers.pdf`

New section for part 4, that explains the purpose of each source file, the main data structures used (or how they were updated), and gives a high-level overview of how the target code is generated.

8 **Ease of grading**

How easy was it for the graders to build your compiler and documentation? Does your compiler work with the grading scripts?

9 **Still works in modes 0, 1, 2, and 3**

We will test earlier modes only using source files that do not generate errors.

8 **Common output**

† 2 `.class` and `.super` lines
† 2 Special method `<init>`
† 2 Java `main()` calls C `main()`
† 2 Java `main()` exits with return code of C `main()`

10 **Code for user functions**

† 3 Correct parameters and return type
† 2 Correct `.method` and `.code` blocks
† 3 Reasonable stack limit
† 2 Correct local count

15 **Function calls and returns**

† 4 Parameter set up
† 4 Correct calls to `lib440` functions
3 Correct calls to user-defined functions
4 Void, char, int, float returns

10 **Expressions: literals, variables**

† 3 Character, integer, and float literals
† 3 Reading local variables and parameters
4 Reading global variables

15 **Operators**

† 10 Binary operators `+`, `-`, `*`, `/`, `%`
5 Unary operators and type conversions

15 **Global variable, local variable, and parameter writes**

3 Local variable initialization
Requires variable initialization support, which was extra credit for parts 2 and 3.
† 4 Assignment expressions with `=`
4 Update assignments: `+=`, `-=`, `*=`, `/=`
4 Pre and post increment and decrement

18 **Arrays**

3 Local array allocation
3 Reading array elements in expressions

	3	Array element assignments with `=`
	3	Array element updates: `+=`, `-=`, `*=`, `/=`
	3	Passing arrays as parameters
	3	Passing string literals as `char[]` parameters
10		**Special method `<clinit>`**
	4	Allocates global arrays
	4	Initializes global variables Requires variable initialization support, which was extra credit for parts 2 and 3.
	2	Method is present when needed, omitted when not needed
4		**Smart stack management** Avoid saving assignment, update, and increment results on the stack for cases where those values will be popped off and discarded.
4		**Missing return statements** Add return statements as needed for void functions. Give an error (with a line number as close as possible to the end of the function body) for non-void functions without a return statement.
4		**Dead code elimination** Statements that can never be reached are eliminated.
100		**Total for students in 440 (max points is 120)**
120		**Total for students in 540 (max points is 140)**

7 Submission

Part	**Penalty applied**
Part 0	40% off
Part 1	30% off
Part 2	20% off
Part 3	10% off

Table 2: Penalty applied when re-grading

Be sure to commit your source code and documentation to your git repository, and to upload (push) those commits to the server so that we may grade them. In Canvas, indicate which parts you would like us to re-grade for reduced credit (see Table 2 for penalty information). Otherwise, we will grade only part 4.