

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 3). 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, error messages should be written to standard error, and your compiler may make a “best effort” to continue processing the input file, or exit.

2 I/O

To manage I/O, for this and the next part of the project, you should assume that our source language has the following *built-in* functions (**i.e., you should add them to your symbol table before compilation starts**).

- `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`.

These functions will be implemented in a supporting library file, `libc.class`, that will be placed in the same directory as your assembled class file.

Most C compilers will correctly compile source inputs that call `getchar()` and `putchar()` without including `stdio.h`; this will allow you to test your compiler against any production C compiler on the same input programs. Alternatively, you can add prototypes for `getchar()` and `putchar()` at the beginning of your test code, assuming your compiler correctly handles multiple prototypes for the same function (as it should).

3 The target language

The target language is Java assembly language, as read by the Krakatau Java assembler (written in Python and available on github at <https://github.com/Storyyeller/Krakatau>). You will build assembly code for a class whose name matches the source file being compiled, with global variables becoming static members, and functions becoming static methods. Your compiler should also produce a proper Java `main()` method, that invokes the static `main()` method from the C source code with prototype

```
int main()
```

and **displays the integer return code from the C `main()`**. (This makes it easier to test your compiler.)

The produced class should be derived from `java/lang/Object`. You will need to output the special method `<init>` to construct an instance of the class, which does nothing except invoke the base constructor. You should also output the special method `<clinit>` to initialize any static members that require initialization: global arrays and global variables that were initialized when declared. If there are no such items to initialize, then method `<clinit>` may be omitted.

Global variables should appear in output before any methods, and in the order in which they are declared. Code implementing each function should appear, in the same order functions are defined, after the special methods but before the Java `main()` method.

Blank lines may be added to improve readability of the output file. Comments, which begin with a semi-colon and end with a newline character, may appear on an empty line or at the end of a valid line. Special comments that begin with a double semi-colon are used to annotate the assembly code for use with a utility that you may use to check your code (see Section 5). You are strongly encouraged to annotate the assembly code with the source code, line number, and a brief description before the generated code; see the examples in Section 4. Section 6 discusses how points are assigned to various code generation features.

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.

4 Examples

4.1 Input: `exprs.c`

```
1  /*
2   These prototypes are not necessary for us in part 4,
3   but allow us to also run this through gcc with no warnings,
4   and to run this through our compiler for part 3.
5  */
6  int getchar();
7  int putchar(int x);
8
9  int main()
10 {
11  /*
12   Read two digits and sum them.
13   No spaces between the digits on input.
14   No error checking, if the two characters
15   are not digits, we get interesting results.
16
17   Also, we do a lot more casting than normal C,
18   in case student compilers don't automatically
19   coerce types from int to char or char to int
20   (this was extra credit in part 3).
21  */
22  char d1, d2, sum;
23  d1 = (char) getchar() - '0';
24  d2 = (char) getchar() - '0';
25  sum = d1 + d2;
26  putchar((int)(d1 + '0'));
27  putchar((int)'+');
28  putchar((int)(d2 + '0'));
29  putchar((int)'=');
30  /*
31   Print the two digit sum, one digit at a time.
32   Leading digit might be zero, but there's no
33   way to suppress that without using if.
34  */
35  putchar((int)(sum / 10 + '0'));
```

```

36  putchar((int)(sum % 10 + '0'));
37  putchar((int)'\n');
38  return sum; /* Produced Java bytecode will show this return value */
39 }

```

4.2 Output: exprs.j

```

1
2 ; Java assembly code
3
4 .class public exprs
5 .super java/lang/Object
6
7 ; Global vars
8
9 .method <init> : ()V
10     .code stack 1 locals 1
11         aload_0
12         invokespecial Method java/lang/Object <init> ()V
13         return
14     .end code
15 .end method
16
17 .method public static main : ()I
18     .code stack 4 locals 3
19         ;; exprs.c 23 expression
20         invokestatic Method libc getchar ()I
21         i2c
22         bipush 48
23         isub
24         i2c
25         dup
26         istore_0 ; store to d1
27         pop
28         ;; exprs.c 24 expression
29         invokestatic Method libc getchar ()I
30         i2c
31         bipush 48
32         isub
33         i2c
34         dup
35         istore_1 ; store to d2
36         pop
37         ;; exprs.c 25 expression
38         iload_0 ; load from d1
39         iload_1 ; load from d2
40         iadd
41         i2c
42         dup
43         istore_2 ; store to sum
44         pop
45         ;; exprs.c 26 expression
46         iload_0 ; load from d1
47         bipush 48

```

```

48      iadd
49      i2c
50      invokestatic Method libc putchar (I)I
51      pop
52      ;; exprs.c 27 expression
53      bipush 43
54      invokestatic Method libc putchar (I)I
55      pop
56      ;; exprs.c 28 expression
57      iload_1 ; load from d2
58      bipush 48
59      iadd
60      i2c
61      invokestatic Method libc putchar (I)I
62      pop
63      ;; exprs.c 29 expression
64      bipush 61
65      invokestatic Method libc putchar (I)I
66      pop
67      ;; exprs.c 35 expression
68      iload_2 ; load from sum
69      bipush 10
70      idiv
71      bipush 48
72      iadd
73      invokestatic Method libc putchar (I)I
74      pop
75      ;; exprs.c 36 expression
76      iload_2 ; load from sum
77      bipush 10
78      irem
79      bipush 48
80      iadd
81      invokestatic Method libc putchar (I)I
82      pop
83      ;; exprs.c 37 expression
84      bipush 10
85      invokestatic Method libc putchar (I)I
86      pop
87      ;; exprs.c 38 return
88      iload_2 ; load from sum
89      ireturn
90  .end code
91  .end method
92
93  .method public static main : ([Ljava/lang/String;)V
94      .code stack 2 locals 2
95      invokestatic Method exprs main ()I
96      istore_1
97      getstatic Field java/lang/System out Ljava/io/PrintStream;
98      ldc 'Return code: '
99      invokevirtual Method java/io/PrintStream print (Ljava/lang/String;)V
100     getstatic Field java/lang/System out Ljava/io/PrintStream;

```

```

101         iload_1
102         invokevirtual Method java/io/PrintStream println (I)V
103         return
104     .end code
105 .end method

```

4.3 Output with smart stack management (extra credit)

```

1
2 ; Java assembly code
3
4 .class public exprs
5 .super java/lang/Object
6
7 ; Global vars
8
9 .method <init> : ()V
10     .code stack 1 locals 1
11         aload_0
12         invokespecial Method java/lang/Object <init> ()V
13         return
14     .end code
15 .end method
16
17 .method public static main : ()I
18     .code stack 4 locals 3
19         ;; exprs.c 23 expression
20         invokestatic Method libc getchar ()I
21         i2c
22         bipush 48
23         isub
24         i2c
25         istore_0 ; store to d1
26         ;; exprs.c 24 expression
27         invokestatic Method libc getchar ()I
28         i2c
29         bipush 48
30         isub
31         i2c
32         istore_1 ; store to d2
33         ;; exprs.c 25 expression
34         iload_0 ; load from d1
35         iload_1 ; load from d2
36         iadd
37         i2c
38         istore_2 ; store to sum
39         ;; exprs.c 26 expression
40         iload_0 ; load from d1
41         bipush 48
42         iadd
43         i2c
44         invokestatic Method libc putchar (I)I
45         pop
46         ;; exprs.c 27 expression

```

```

47     bipush 43
48     invokestatic Method libc putchar (I)I
49     pop
50     ;; exprs.c 28 expression
51     iload_1 ; load from d2
52     bipush 48
53     iadd
54     i2c
55     invokestatic Method libc putchar (I)I
56     pop
57     ;; exprs.c 29 expression
58     bipush 61
59     invokestatic Method libc putchar (I)I
60     pop
61     ;; exprs.c 35 expression
62     iload_2 ; load from sum
63     bipush 10
64     idiv
65     bipush 48
66     iadd
67     invokestatic Method libc putchar (I)I
68     pop
69     ;; exprs.c 36 expression
70     iload_2 ; load from sum
71     bipush 10
72     irem
73     bipush 48
74     iadd
75     invokestatic Method libc putchar (I)I
76     pop
77     ;; exprs.c 37 expression
78     bipush 10
79     invokestatic Method libc putchar (I)I
80     pop
81     ;; exprs.c 38 return
82     iload_2 ; load from sum
83     ireturn
84 .end code
85 .end method
86
87 .method public static main : ([Ljava/lang/String;)V
88     .code stack 2 locals 2
89     invokestatic Method exprs main ()I
90     istore_1
91     getstatic Field java/lang/System out Ljava/io/PrintStream;
92     ldc 'Return code: '
93     invokevirtual Method java/io/PrintStream print (Ljava/lang/String;)V
94     getstatic Field java/lang/System out Ljava/io/PrintStream;
95     iload_1
96     invokevirtual Method java/io/PrintStream println (I)V
97     return
98 .end code
99 .end method

```

5 Checking your generated code

Ultimately, you should be able to assemble the code generated by your compiler (using the Krakatau assembler) to obtain a class file. You can then run this class file, just as if it were compiled from Java source. The script `Run.sh` is based on this idea:

1. It first runs your compiler with mode `-4` on the C source code. If the instructor solution generates an error message, then the script checks that your compiler generated an error message on the same line.
2. Otherwise, the script runs the assembler on your compiler's output.
3. The script runs the resulting `.class` file on a JVM, with one or more input files (in case the C source calls `getchar()`) and checks the output.

You will need to implement methods `putchar()` and `getchar()`, but this may be done using Java source `libc.java`:

```
1  import java.io.IOException;
2
3  class libc {
4      public static int putchar(int c) {
5          System.out.print((char) c);
6          return c;
7      }
8      public static int getchar() throws IOException {
9          return System.in.read();
10     }
11 };
```

It is recommended that you examine the assembly code produced by your compiler. For longer or more complex code (without any branching or loops), you might want to use the `jexpr` utility to examine your code and produce a somewhat more readable summary of what each function computes. Running `jexpr` on the code shown in Section 4.3 produces the following.

```
1  V <init> ( )
2  {
3      calling local0.<init>()
4      ;
5      ; Requested stack size 1
6      ; Required stack size 1
7      ; 0 items remaining on the stack
8      ;
9  }
10
11 I main ( )
12 {
13     ;; exprs.c 23 expression
14     calling libc::getchar()
15     local0 = i2c(i2c(libc::getchar()) - 48)
16     ;; exprs.c 24 expression
17     calling libc::getchar()
18     local1 = i2c(i2c(libc::getchar()) - 48)
19     ;; exprs.c 25 expression
20     local2 = i2c(local0 + local1)
21     ;; exprs.c 26 expression
22     calling libc::putchar(i2c(local0 + 48))
23     pop libc::putchar(i2c(local0 + 48))
```

```

24     ;; exprs.c 27 expression
25     calling libc::putchar(43)
26     pop libc::putchar(43)
27     ;; exprs.c 28 expression
28     calling libc::putchar(i2c(local1 + 48))
29     pop libc::putchar(i2c(local1 + 48))
30     ;; exprs.c 29 expression
31     calling libc::putchar(61)
32     pop libc::putchar(61)
33     ;; exprs.c 35 expression
34     calling libc::putchar(local2 / 10 + 48)
35     pop libc::putchar(local2 / 10 + 48)
36     ;; exprs.c 36 expression
37     calling libc::putchar(local2 % 10 + 48)
38     pop libc::putchar(local2 % 10 + 48)
39     ;; exprs.c 37 expression
40     calling libc::putchar(10)
41     pop libc::putchar(10)
42     ;; exprs.c 38 return
43     ireturn local2
44     ;
45     ; Requested stack size 4
46     ; Required stack size 2
47     ; 0 items remaining on the stack
48     ;
49 }
50
51 V main ( [Ljava/lang/String; )
52 {
53     calling exprs::main()
54     local1 = exprs::main()
55     calling java/lang/System.out.print('Return code: ')
56     calling java/lang/System.out.println(local1)
57     ;
58     ; Requested stack size 2
59     ; Required stack size 2
60     ; 0 items remaining on the stack
61     ;
62 }

```

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.	
10	Ease of grading	
4	Building	
	How easy was it for the graders to build your compiler and documentation? For full credit, simply running “make” should build both the documentation and the compiler executable, and running “make clean” should remove all generated files.	
6	Output and formatting	
	Does the -o switch work? Is your output formatted correctly? Are other messages written to standard error?	
10	Still works in modes 0, 1, 2, and 3	
10	Always present output	
†	3	Class with proper name; super
†	3	Special method <init>
†	4	Java main(), calls C main() and shows return value
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
	3	Function call
†	4	Correct calls to built-ins getchar and putchar
	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 initialization
	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 <code>char[]</code> parameters
10		Special method <clinit>
	4	Initializes 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
5		Smart stack management
		Avoid using the stack for top-level expressions, when those values are ultimately going to be popped off and discarded.
<hr/>		
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.