COM S 440/540 Project part 5

Code generation: control flow

1 Requirements for part 5

When executed with a mode of 5, 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 (still Java assembly). For this part of the project, your compiler must generate correct code for expressions (from part 4), and for (possibly nested) branching statements and loops. 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. Note: unlike for part 4, the Java `main()` function should simply call the C `main()` function.

2 Examples

2.1 Input: `fib.c`

```
1  /*
2    Computes and prints the first 20 Fibonacci numbers.
3  */
4
5  int putchar(int c); /* In stdio.h */
6
7  void print(int x)
8  {
9    if (x<0) {
10     putchar(45);
11     print(-x);
12     return;
13   }
14   if (x>9) {
15     print(x/10);
16   }
17   x = x % 10;
18   putchar(48+x);
19 }
20
21 void print_two(int x)
22 {
23   if (x>9) {
24     putchar(48 + x/10);
25   } else {
26     putchar(32);
27   }
28   putchar(48 + x % 10);
29   putchar(32);
```

```
  putchar(58);
  putchar(32);
}

int main()
{
  int i, f1, f2, f3;
  f1 = 0;
  f2 = 1;
  print_two(0);
  print(0);
  putchar(10);
  i=20;
  while (i) {
    print_two(20 - --i);
    print(f2);
    putchar(10);
    f3 = f1 + f2;
    f1 = f2;
    f2 = f3;
  }
  return 0;
}
```

2.2 Output: `fib.j`

```
; Java assembly code

.class public fib
.super java/lang/Object

; Global vars

.method <init> : ()V
   .code stack 1 locals 1
      aload_0
      invokespecial Method java/lang/Object <init> ()V
      return
   .end code
.end method

.method public static print : (I)V
   .code stack 4 locals 1
      ;; fib.c 9 if
      ; empty else
      iload_0 ; load from x
      iconst_0
      if_icmpge L1
      ;; fib.c 10 expression
      bipush 45
      invokestatic Method libc putchar (I)I
      pop
      ;; fib.c 11 expression
```

```
29          iload_0 ; load from x
30          ineg
31          invokestatic Method fib print (I)V
32          ;; fib.c 12 return
33          return
34      L1:
35          ;; fib.c 14 if
36          ; empty else
37          iload_0 ; load from x
38          bipush 9
39          if_icmple L2
40          ;; fib.c 15 expression
41          iload_0 ; load from x
42          bipush 10
43          idiv
44          invokestatic Method fib print (I)V
45      L2:
46          ;; fib.c 17 expression
47          iload_0 ; load from x
48          bipush 10
49          irem
50          istore_0 ; store to x
51          ;; fib.c 18 expression
52          bipush 48
53          iload_0 ; load from x
54          iadd
55          invokestatic Method libc putchar (I)I
56          pop
57          return ; implicit return
58      .end code
59  .end method
60
61  .method public static print_two : (I)V
62      .code stack 3 locals 1
63          ;; fib.c 23 if
64          iload_0 ; load from x
65          bipush 9
66          if_icmple L1
67          ;; fib.c 24 expression
68          bipush 48
69          iload_0 ; load from x
70          bipush 10
71          idiv
72          iadd
73          invokestatic Method libc putchar (I)I
74          pop
75          goto L2
76      L1:
77          ;; fib.c 26 expression
78          bipush 32
79          invokestatic Method libc putchar (I)I
80          pop
81      L2:
```

```
82          ;; fib.c 28 expression
83          bipush 48
84          iload_0 ; load from x
85          bipush 10
86          irem
87          iadd
88          invokestatic Method libc putchar (I)I
89          pop
90          ;; fib.c 29 expression
91          bipush 32
92          invokestatic Method libc putchar (I)I
93          pop
94          ;; fib.c 30 expression
95          bipush 58
96          invokestatic Method libc putchar (I)I
97          pop
98          ;; fib.c 31 expression
99          bipush 32
100         invokestatic Method libc putchar (I)I
101         pop
102         return ; implicit return
103     .end code
104 .end method
105
106 .method public static main : ()I
107     .code stack 4 locals 4
108         ;; fib.c 37 expression
109         iconst_0
110         istore_1 ; store to f1
111         ;; fib.c 38 expression
112         iconst_1
113         istore_2 ; store to f2
114         ;; fib.c 39 expression
115         iconst_0
116         invokestatic Method fib print_two (I)V
117         ;; fib.c 40 expression
118         iconst_0
119         invokestatic Method fib print (I)V
120         ;; fib.c 41 expression
121         bipush 10
122         invokestatic Method libc putchar (I)I
123         pop
124         ;; fib.c 42 expression
125         bipush 20
126         istore_0 ; store to i
127         ;; fib.c 43 while
128     L1:
129         iload_0 ; load from i
130         ifeq L2
131         ;; fib.c 44 expression
132         bipush 20
133         iinc 0 -1
134         iload_0 ; load from i
```

```
135         isub
136         invokestatic Method fib print_two (I)V
137         ;; fib.c 45 expression
138         iload_2 ; load from f2
139         invokestatic Method fib print (I)V
140         ;; fib.c 46 expression
141         bipush 10
142         invokestatic Method libc putchar (I)I
143         pop
144         ;; fib.c 47 expression
145         iload_1 ; load from f1
146         iload_2 ; load from f2
147         iadd
148         istore_3 ; store to f3
149         ;; fib.c 48 expression
150         iload_2 ; load from f2
151         istore_1 ; store to f1
152         ;; fib.c 49 expression
153         iload_3 ; load from f3
154         istore_2 ; store to f2
155         goto L1
156     L2:
157         ;; fib.c 51 return
158         iconst_0
159         ireturn
160     .end code
161 .end method
162
163 .method public static main : ([Ljava/lang/String;)V
164     .code stack 1 locals 1
165         invokestatic Method fib main ()I
166         pop
167         return
168     .end code
169 .end method
```

3 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 `-5` 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.

As with part 4, you will need to implement methods `putchar()` and `getchar()` in `libc.class`, but this may be done using Java source.

4 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, and a deficit of points will impact your overall grade. Excess points will count as extra credit.

For code generation "without short circuiting", your compilers will be tested using integer variables for the condition. The basic tests for each construct are as follows, where `x` is an integer variable.

```
if (x) { /* statements */ }

if (x) { /* statements */ } else { /* statements */ }

while (x) { /* statements */ }

do { /* statements */ } while (x);

for (/* initialize */; x; /* update */) { /* statements */ }
```

More advanced tests will use a single comparison as the condition.

Tests for short-circuiting boolean expressions will call functions that output characters (so make sure those are working) as part of the boolean expression, to make sure the expression short circuits.

Points	Description
15	**Documentation**
3	`README.txt` How to build the compiler and documentation. Updated to show which part 5 features are implemented.
12	`developers.pdf` New section for part 5, 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**
15	**Expressions and function calls** This includes the most basic functionality from the previous part of the project, namely function calls and assignments to variables, that will be necessary to test this part of the project.
55	**Without short circuiting**
5	if–then
5	if–then–else
5	while

5	do–while
8	for
5	ternary operator `?:`
5	`break` (requires a working loop)
5	`continue` (requires a working loop)
12	comparisons: `==`, `!=`, `>`, `>=`, `<`, `<=`
30	**With short circuiting**
5	and, or, not
5	comparisons: `==`, `!=`, `>`, `>=`, `<`, `<=`
5	Boolean assignments
5	if–then, if–then–else, ternary operator
5	while, do–while
5	for
100	**Total for students in 440 (max points is 120)**
120	**Total for students in 540**

5 Submission

Part	Penalty applied
Part 0	50% off
Part 1	40% off
Part 2	30% off
Part 3	20% off
Part 4	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 5.