```java
/*
 * Ryan Long
 * CS 4100 Compiler Project Part 1: Foundations
 * 9/18/2020
 *
 * This project creates a SYmbol Table, Quad Table, and Reserve Table
 * and uses them to carry out instructions through an interpreter
 * and generate output for a Factorial(10) and Summation(10) function.
 * A traceOn boolean is used to trace output if desired.
 */

import java.util.ArrayList;

////////////
//Classes /
//////////

class Symbol {

    private String name;
    private int kind;

    private String stringValue;
    private int intValue;
    private double doubleValue;

// Overloaded Symbol objects for int, double, or String values
    public Symbol(String name, int kind, String value) {
        this.name = name;
        this.kind = kind;
        stringValue = value;
    }

    public Symbol(String name, int kind, int value) {
        this.name = name;
        this.kind = kind;
        intValue = value;
    }

    public Symbol(String name, int kind, double value) {
        this.name = name;
        this.kind = kind;
```

```
43          doubleValue = value;
44      }
45
46  // Getters
47      public String getName() { return name; }
48      public int getKind() { return kind; }
49
50  // Overloaded Getters to return a specific value type
51      public String getStringValue() { return stringValue; }
52      public int getIntValue() { return intValue; }
53      public double getDoubleValue() { return doubleValue; }
54
55  // Setters
56      public void setName(String name) { stringValue = name; }
57      public void setKind(int kind) { this.kind = kind; }
58      public void setStringValue(String value) { stringValue = value; }
59      public void setIntValue(int value) { intValue = value; }
60      public void setDoubleValue(double value) { doubleValue = value; }
61
62  @Override
63      public String toString() {
64      switch(kind) {
65          case 0:
66              return name + "\t" + kind + "\t" + stringValue;
67          case 1:
68              return name + "\t" + kind + "\t" + intValue;
69          case 2:
70              return name + "\t" + kind + " \t" + stringValue;
71          default:
72              return "DNE";
73          }
74      }
75  } // class Symbol
76
77  class SymbolTable {
78
79      ArrayList<Symbol> symbList = new ArrayList<Symbol>();
80
81  // Constructor that builds an empty Symbol Table
82  // Format is (name, kind, value)
83  // kind 0 is String value, 1 is int value, 2 is double value
84      public SymbolTable() {
```

```
85      }
86
87  //Adds symbol with given kind and value (name) to the symbol table,
88  //automatically setting the correct data_type, and returns to
89  //the index where the symbol was located.
90  //If symbol is already in table, make no change, return to index where symbol was found.
91      public int AddSymbol(String symbol, int kind, String value) {
92          symbList.add(new Symbol(symbol, kind, value));
93          return symbList.size() - 1;
94      }
95
96      public int AddSymbol(String symbol, int kind, int value) {
97          symbList.add(new Symbol(symbol, kind, value));
98          return symbList.size() - 1;
99      }
100
101     public int AddSymbol(String symbol, int kind, double value) {
102         symbList.add(new Symbol(symbol, kind, value));
103         return symbList.size() - 1;
104     }
105
106 //Returns the index where symbol is found, or -1 if not in table
107     public int LookupSymbol(String symbol) {
108         for (int i = 0; i < symbList.size(); i++) {
109             if (symbList.get(i).getName() == symbol)
110                 return i;
111         }
112         return -1;
113     }
114
115 //Return kind, data type, and value fields stored at index
116     public Symbol GetSymbol(int index) {
117         return symbList.get(index);
118     }
119
120 //Set appropriate fields at slot indicated by index
121     public void UpdateSymbol(int index, int kind, int value) {
122         GetSymbol(index).setIntValue(value);
123     }
124
125     public void UpdateSymbol(int index, int kind, double value) {
126         GetSymbol(index).setDoubleValue(value);
```

```
127     }
128
129     public void UpdateSymbol(int index, int kind, String value) {
130         GetSymbol(index).setStringValue(value);
131     }
132
133 //Prints the utilized rows of the symbol table in a neat tabular format,
134 //showing only the value fields which is active for that row
135     public void PrintSymbolTable() {
136         System.out.println("\n======== Initial Symbol Table Contents ========");
137         System.out.println("Name  | Kind  | Value");
138
139         for (int i = 0; i < symbList.size(); i++)
140             System.out.println(symbList.get(i));    // call to toString method
141     }
142 }
143
144 //QuadTable creates a new, empty QuadTable ready for data to be added, with the specified number of rows (size)
145 class QuadTable {
146
147     static int[][] quadArr;
148
149     public void Initialize(int rows, ReserveTable R) {
150         int cols = 4;
151         quadArr = new int[rows][cols];
152
153 //populate arrray with -1 values
154         for (int i = 0; i < rows; i++)
155             for (int j = 0; j < cols; j++)
156                 quadArr[i][j] = -1;
157     }
158
159 //Return the int index of the next open slot in QuadTable.
160     public int NextQuad() {
161         for (int i = 0; i < quadArr.length; i++) {
162             if (quadArr[i][0] == -1) {
163                 return i;
164             }
165         }
166         //else, array is full
167         return 0;
168     }
```

```
169
170 //Expands the active length of the QuadTable by adding a new row
171 //at the NextQuad slot, with the parameter sent as the new contents,
172 //and increments the NextQuad counter to the next available (empty) index
173     public void AddQuad(int opcode, int op1, int op2, int op3, int next) {
174         quadArr[next][0] = opcode;
175         quadArr[next][1] = op1;
176         quadArr[next][2] = op2;
177         quadArr[next][3] = op3;
178     }
179
180 //Returns the data for the opcode and three operands located at index
181     public static int[] GetQuad(int index) {
182         int[] quadRow = quadArr[index];
183         return quadRow;
184     }
185
186 //Changes the contents of the existing quad at index.
187 //Used only when backfilling jump addresses later during code generation
188     public void SetQuad(int index, int opcode, int op1, int op2, int op3) {
189     }
190
191 //Returns the mnemonic string ('ADD', 'PRINT', etc) associated with the opcode parameter.
192     public String GetMnemonic(int opcode, ReserveTable R) {
193         return R.LookupCode(opcode);
194     }
195
196 //Prints the currently used contents of the Quad table in neat tabular format
197     public void PrintQuadTable(ReserveTable R) {
198         int firstColIndex = 0;
199         System.out.println("\n======== Quad Table contents ========");
200         System.out.println("Opcode | op1  | op2 |  op3");
201         System.out.println("-------------------------");
202
203         for (int i = 0; i < quadArr.length; i++) { // Loop through rows
204             for (int j = 0; j < quadArr[i].length; j++) { // Loop through elements in the row
205
206                 // Determine if Opcode ([0][0], [1][0], [2][0], [3][0], etc)
207                 if (i == firstColIndex) {
208                     System.out.print(R.LookupCode(quadArr[i][j]) + "\t "); // If in the first column, print the Opcode string
209                     firstColIndex++;
210                 } else
```

```
                    System.out.print(quadArr[i][j] + "\t"); // Else, print the op1, op2, or op3 value
                } // inner for
                System.out.println();
            } // outer for
        }
}

//ReserveTable is a lookup table ADT used for the opcode lookup, and later in the compiler,
//a separate instance will hold the reserved word list for the language.
//Each indexed entry is a pair consisting of a name string and integer code.
//The table as we use it is static, and initialized once at the start of the program,
//and then only used for lookups later on
class ReserveTable {

    ArrayList<String> resArr;

    //Constructor, as needed
    ReserveTable() {
        resArr = new ArrayList<>();
    }

//Returns the index of the row where the data was placed, just adds to end of list
    public int Add(String name, int code) {
        resArr.add(name);
        resArr.set(code, name);
        return code;
    }

//Returns the code associated with name if name is in the table, else return -1
    public int LookupName(String name) {
        for (int i = 0; i < resArr.size(); i++) {
            if (resArr.get(i) == name) {
                return i;
            }
        }
        return -1;
    }

//Returns the associated name if code is there, else return an empty string
    public String LookupCode(int code) {
        if (code <= resArr.size())
            return resArr.get(code);
```

```
253         return "";
254     }
255 
256 //Prints the currently used contents of the Reserve table in neat tabular format
257     public void PrintReserveTable() {
258         System.out.println("\n======== Reserve Table Contents ========");
259         for (int i = 0; i < (resArr.size() - 1); i++) {
260             System.out.print(i + " " + resArr.get(i) + ", ");
261 
262             if (i == 8)                 // Split the printout into 2 rows
263                 System.out.println();
264         }
265         System.out.println(resArr.size() - 1 + " " + resArr.get(resArr.size() - 1));  // Print last item without a comma
266     }
267 }
268 
269 public class Compilers_v3 {
270 
271 //////////
272 // MAIN //
273 //////////
274     public static void main(String[] args) {
275 
276         boolean traceOn = true;
277 
278         // Create a Reserve Table, Symbol Table, and Quad Table
279         ReserveTable R = new ReserveTable();
280         QuadTable Q = new QuadTable();
281         SymbolTable S = new SymbolTable();
282 
283         BuildSymbolTable(S); // Allocate Symbols to the SymbolTable and print them
284 
285         BuildQuads(R, Q); // Build the QuadTable and print it
286 
287         // Run the interpreter with traceOn, showing step-by-step instructions
288         InterpretQuads(Q, R, S, traceOn);
289 
290         // Run the interpreter without traceOn, only printing the final output
291         InterpretQuads(Q, R, S, traceOn = false);
292 
293     } // main
294 
```

```
295 //////////////
296 //Functions /
297 //////////////
298
299     public static void InterpretQuads(QuadTable Q, ReserveTable R, SymbolTable S, boolean traceOn) {
300
301         int pc = 0;
302         final int MAXQUAD = 1000;
303         int[] quadRow;
304         boolean stopFlag = false;
305         String instruction;         // The Opcode (MOV, ADD, etc)
306
307         if (traceOn) System.out.println("\nRunning Summation(10) with trace enabled:"); /* Set to Summation when running
    Summation function*/
308         else System.out.println("\nRunning Summation(10) without trace:"); /* Set to Summation when running Summation
    function*/
309         while (pc < MAXQUAD) {
310             String pcPadded = String.format("%03d", pc);
311             if (traceOn) System.out.printf("\nPC = %s:", pcPadded);
312
313             quadRow = (Q.GetQuad(pc));
314             int q1, q2, q3;
315
316             for (int i = 0; i < quadRow.length; i++) {  // Loop through the current quadcode row
317                 instruction = Q.GetMnemonic(quadRow[i], R);
318
319                 if (i == 0) {
320                     if (traceOn) System.out.print(" " + instruction + "\t"); // Print the Opcode
321                     if (traceOn) System.out.print(quadRow[1] + "\s" + quadRow[2] + "\s" + quadRow[3]);
322
323                     q1 = quadRow[1];    // Assign each element in the row to a variable for greater readability
324                     q2 = quadRow[2];
325                     q3 = quadRow[3];
326
327                     // Instructions for instructions
328                     switch (instruction) {
329
330                     // TERMINATE
331                     case "STOP":            // If STOP call is found, halt the while loop after printing the remaining 0 0 0
    opcodes
332                         stopFlag = true;
333                         break;
```

```
334
335                 // MATH
336                 case "DIV": // Compute op1 / op2, place result into op3
337                     S.UpdateSymbol(q1, 0, S.GetSymbol(q2).getIntValue() / S.GetSymbol(q1).getIntValue());
338                     if (traceOn) printMath(q1, S);
339                     break;
340                 case "MUL": // Compute op1 * op2, place result into op3
341                     S.UpdateSymbol(q1, 0, S.GetSymbol(q2).getIntValue() * S.GetSymbol(q1).getIntValue());
342                     if (traceOn) printMath(q1, S);
343                     break;
344                 case "SUB": // Compute op1 - op2, place result into op3
345                     S.UpdateSymbol(q3, 0, S.GetSymbol(q1).getIntValue() - S.GetSymbol(q2).getIntValue());
346                     if (traceOn) printMath(q3,S);
347                     break;
348                 case "ADD": // Compute op1 + op2, place result into op3
349                     S.UpdateSymbol(q1, 0, S.GetSymbol(q2).getIntValue() + S.GetSymbol(q3).getIntValue());
350                     if (traceOn) printMath(q1, S);
351                     break;
352
353                 // DATA STORAGE
354                 case "MOV": // Copy op1 into op3
355                     S.UpdateSymbol(q3, 0, S.GetSymbol(q1).getIntValue());
356                     if (traceOn) printMov(q3, S);
357                     break;
358                 case "STI": // Copy op1 into op2 + offset op3
359                     S.UpdateSymbol(q1, 0, S.GetSymbol(q2 + q3).getIntValue());
360                     break;
361                 case "LDI": // Copy op1 + offset op2 into op3
362                     S.UpdateSymbol(q3, 0, S.GetSymbol(q1 + q2).getIntValue());
363                     break;
364
365                 // BRANCH INSTRUCTIONS
366                 case "BNZ": // if op1 not 0, pc = op3
367                     if (S.GetSymbol(q1).getIntValue() != 0)
368                         pc = q3 - 1;
369                     break;
370                 case "BNP": // if op1 less or equal to 0, pc = op3
371                     if (S.GetSymbol(q1).getIntValue() <= 0)
372                         pc = q3 - 1;
373                     break;
374                 case "BNN": // if op1 greater or equal to 0, pc = op3
375                     if (S.GetSymbol(q1).getIntValue() >= 0)
```

```
376                         pc = q3 - 1;
377                     break;
378                 case "BZ": // if op1 is 0, pc = op3
379                     if (S.GetSymbol(q1).getIntValue() == 0)
380                         pc = q3 - 1;
381                     break;
382                 case "BP": // if op1 is greater than 0, pc = op3
383                     if (S.GetSymbol(q1).getIntValue() > 0)
384                         pc = q3 - 1;
385                     break;
386                 case "BN": // If op1 < 0, set program counter equal to op3
387                     if (S.GetSymbol(q1).getIntValue() < 0) {
388                         if (traceOn) System.out.print("\t-----> " + S.GetSymbol(q1).getIntValue() + " < 0, branch to pc " +
    q3);
389                         pc = q3 - 1;
390                     }
391                     else
392                         if (traceOn) System.out.print("\t-----> " + S.GetSymbol(q1).getIntValue() + " not < 0, continue");
393                     break;
394                 case "BR": // Set program counter equal to op3
395                     if (traceOn) System.out.print("\t-----> Unconditional Branch, setting pc to " + q3);
396                     pc = q3 - 1;
397                     break;
398 
399                 // UTILITY
400                 case "PRINT": // Write Symbol Table name and value of op1
401                     if (traceOn) System.out.print("\t****** Summation is " + S.GetSymbol(q3).getIntValue() + " ******");
402                     else System.out.println("Summation is " + S.GetSymbol(q3).getIntValue());
403                     break;
404 
405                 default:
406                     break;
407                 } // case
408             } // if
409         } // for
410 
411         if (stopFlag == true)
412             break;
413         pc += 1;
414     } // while
415     if (traceOn) System.out.print("\t-----> Program Terminated\n\n");
416 }
```

```
417
418     public static void BuildQuads(ReserveTable R, QuadTable Q) {
419
420 // Populate Reserve Table
421         R.Add("STOP", 0);   R.Add("DIV", 1); R.Add("MUL", 2); R.Add("SUB", 3); R.Add("ADD", 4);
422         R.Add("MOV", 5);    R.Add("STI", 6); R.Add("LDI", 7); R.Add("BNZ", 8); R.Add("BNP", 9);
423         R.Add("BNN", 10);   R.Add("BZ", 11); R.Add("BP", 12); R.Add("BN", 13); R.Add("BR", 14);
424         R.Add("BINDR", 15); R.Add("PRINT", 16);
425
426         R.PrintReserveTable();
427
428 // Populate Quad Table
429         int quadTableSize = 11;
430         Q.Initialize(quadTableSize, R); // create QuadTable with <size> rows
431
432         Q.AddQuad(5, 4, 0, 0, Q.NextQuad());
433         Q.AddQuad(5, 5, 0, 1, Q.NextQuad());
434         Q.AddQuad(5, 5, 0, 2, Q.NextQuad());
435         Q.AddQuad(3, 0, 2, 6, Q.NextQuad());
436         Q.AddQuad(13, 6, 0, 8, Q.NextQuad());
437         Q.AddQuad(2, 1, 2, 1, Q.NextQuad());   /* Set first value to 2 (MUL) for Factorial, 4 (ADD) for Summation*/
438         Q.AddQuad(4, 2, 5, 2, Q.NextQuad());
439         Q.AddQuad(14, 0, 0, 3, Q.NextQuad());
440         Q.AddQuad(5, 1, 0, 3, Q.NextQuad());
441         //Q.AddQuad(3, 3, 5, 3, Q.NextQuad()); /* Subtract 1 from final result for Summation*/
442         Q.AddQuad(16, 0, 0, 3, Q.NextQuad());
443         Q.AddQuad(0, 0, 0, 0, Q.NextQuad());
444
445         Q.PrintQuadTable(R);
446     }
447
448     public static void BuildSymbolTable(SymbolTable S) {
449
450         S.AddSymbol("n", 1, 0);
451         S.AddSymbol("prod", 1, 0);
452         S.AddSymbol("count", 1, 0);
453         S.AddSymbol("fact", 1, 0);
454         S.AddSymbol("10", 1, 10);           /* Change this value to set n */
455         S.AddSymbol("1", 1, 1);
456         S.AddSymbol("temp", 1, 0);
457         S.AddSymbol("sub", 1, 2);           /* Exists to determine correct value of Summation*/
458
```

```
459         S.PrintSymbolTable();
460     }
461
462 // printMath and other print<Foo> functions are called if traceOn is true,
463 // and exist to keep the interpreter free of excess clutter
464     public static void printMath(int q1, SymbolTable S) {
465         System.out.print("\t-----> " + S.GetSymbol(q1).getName() + " set to " + S.symbList.get(q1).getIntValue());
466     }
467
468     public static void printMov(int q3, SymbolTable S) {
469         System.out.print("\t-----> " + S.GetSymbol(q3).getName() + " set to " + S.symbList.get(q3).getIntValue());
470     }
471
472 } // Compilers_v3
473
474
```

Output

Factorial(10) Tables:

```
======== Initial Symbol Table Contents ========
Name   | Kind   | Value
n        1        0
prod     1        0
count    1        0
fact     1        0
10       1        10
1        1        1
temp     1        0

======== Reserve Table Contents ========
0 STOP, 1 DIV, 2 MUL, 3 SUB, 4 ADD, 5 MOV, 6 STI, 7 LDI, 8 BNZ,
9 BNP, 10 BNN, 11 BZ, 12 BP, 13 BN, 14 BR, 15 BINDR, 16 PRINT

======== Quad Table contents ========
Opcode | op1   | op2 |   op3
--------------------------
MOV      4      0        0
MOV      5      0        1
MOV      5      0        2
SUB      0      2        6
BN       6      0        8
MUL      1      2        1
ADD      2      5        2
BR       0      0        3
MOV      1      0        3
PRINT    0      0        3
STOP     0      0        0
```

Factorial(10) trace enabled:

Left column is the start of output, right column is the end of output

```
Running Factorial(10) with trace enabled:

PC = 000: MOV   4 0 0   -----> n set to 10
PC = 001: MOV   5 0 1   -----> prod set to 1
PC = 002: MOV   5 0 2   -----> count set to 1
PC = 003: SUB   0 2 6   -----> temp set to 9
PC = 004: BN    6 0 8   -----> 9 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 1
PC = 006: ADD   2 5 2   -----> count set to 2
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 8
PC = 004: BN    6 0 8   -----> 8 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 2
PC = 006: ADD   2 5 2   -----> count set to 3
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 7
PC = 004: BN    6 0 8   -----> 7 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 6
PC = 006: ADD   2 5 2   -----> count set to 4
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 6
PC = 004: BN    6 0 8   -----> 6 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 24
PC = 006: ADD   2 5 2   -----> count set to 5
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 5
PC = 004: BN    6 0 8   -----> 5 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 120
PC = 006: ADD   2 5 2   -----> count set to 6
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 4
PC = 004: BN    6 0 8   -----> 4 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 720
PC = 006: ADD   2 5 2   -----> count set to 7
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 3
PC = 004: BN    6 0 8   -----> 3 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 5040
PC = 006: ADD   2 5 2   -----> count set to 8
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 2
PC = 004: BN    6 0 8   -----> 2 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 40320
PC = 006: ADD   2 5 2   -----> count set to 9
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 1
PC = 004: BN    6 0 8   -----> 1 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 362880
PC = 006: ADD   2 5 2   -----> count set to 10
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 0
PC = 004: BN    6 0 8   -----> 0 not < 0, continue
PC = 005: MUL   1 2 1   -----> prod set to 3628800
PC = 006: ADD   2 5 2   -----> count set to 11
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to -1
PC = 004: BN    6 0 8   -----> -1 < 0, branch to pc 8
PC = 008: MOV   1 0 3   -----> fact set to 3628800
PC = 009: PRINT 0 0 3   ****** Factorial is 3628800 ******
PC = 010: STOP  0 0 0   -----> Program Terminated
```

Factorial(10) trace disabled:

```
Running Factorial(10) without trace:
Factorial is 3628800
```

Summation(10) Tables:

```
======== Initial Symbol Table Contents ========
Name  | Kind  | Value
n       1       0
prod    1       0
count   1       0
fact    1       0
10      1       10
1       1       1
temp    1       0
sub     1       2

======== Reserve Table Contents ========
0 STOP, 1 DIV, 2 MUL, 3 SUB, 4 ADD, 5 MOV, 6 STI, 7 LDI, 8 BNZ,
9 BNP, 10 BNN, 11 BZ, 12 BP, 13 BN, 14 BR, 15 BINDR, 16 PRINT

======== Quad Table contents ========
Opcode | op1  | op2 |  op3
--------------------------
MOV      4      0       0
MOV      5      0       1
MOV      5      0       2
SUB      0      2       6
BN       6      0       8
ADD      1      2       1
ADD      2      5       2
BR       0      0       3
MOV      1      0       3
SUB      3      5       3
PRINT    0      0       3
STOP     0      0       0
```

Summation(10) trace enabled:

Left column is the start of output, right column is the end of output

```
Running Summation(10) with trace enabled:

PC = 000: MOV   4 0 0   -----> n set to 10
PC = 001: MOV   5 0 1   -----> prod set to 1
PC = 002: MOV   5 0 2   -----> count set to 1
PC = 003: SUB   0 2 6   -----> temp set to 9
PC = 004: BN    6 0 8   -----> 9 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 2
PC = 006: ADD   2 5 2   -----> count set to 2
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 8
PC = 004: BN    6 0 8   -----> 8 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 4
PC = 006: ADD   2 5 2   -----> count set to 3
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 7
PC = 004: BN    6 0 8   -----> 7 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 7
PC = 006: ADD   2 5 2   -----> count set to 4
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 6
PC = 004: BN    6 0 8   -----> 6 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 11
PC = 006: ADD   2 5 2   -----> count set to 5
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 5
PC = 004: BN    6 0 8   -----> 5 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 16
PC = 006: ADD   2 5 2   -----> count set to 6
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 4
PC = 004: BN    6 0 8   -----> 4 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 22
PC = 006: ADD   2 5 2   -----> count set to 7
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 3
PC = 004: BN    6 0 8   -----> 3 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 29
PC = 006: ADD   2 5 2   -----> count set to 8
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 2
PC = 004: BN    6 0 8   -----> 2 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 37
PC = 006: ADD   2 5 2   -----> count set to 9
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 1
PC = 004: BN    6 0 8   -----> 1 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 46
PC = 006: ADD   2 5 2   -----> count set to 10
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to 0
PC = 004: BN    6 0 8   -----> 0 not < 0, continue
PC = 005: ADD   1 2 1   -----> prod set to 56
PC = 006: ADD   2 5 2   -----> count set to 11
PC = 007: BR    0 0 3   -----> Unconditional Branch, setting pc to 3
PC = 003: SUB   0 2 6   -----> temp set to -1
PC = 004: BN    6 0 8   -----> -1 < 0, branch to pc 8
PC = 008: MOV   1 0 3   -----> fact set to 56
PC = 009: SUB   3 5 3   -----> fact set to 55
PC = 010: PRINT 0 0 3   ****** Summation is 55 ******
PC = 011: STOP  0 0 0   -----> Program Terminated
```

Summation(10) trace disabled:

```
Running Summation(10) without trace:
Summation is 55
```