Title

Author1 Author2

January 13, 2022

Abstract

This is all about…

1 Eliminating Goto Statements from COBOL

1.1 Summary of *Taming Control Flow*

Taming Control Flow: A Structured Approach to Eliminating Goto Statements formalizes a procedure to eliminate all goto statements from a program. This is done by making the source conform to specific standards of structured programming. There are two categories of transformations that must be done to eliminate all goto statements from a program. These categories include:

- **Eliminating the goto statement**, and
- **Moving the goto statement** in preparation for elimination.

Only a subset of the transformations described in each category is relevant for eliminating gotos from COBOL programs. We will discuss the reasons for this shortly, but first we will give some important definitions (with examples) that will be used to define the transformations.

1.1.1 Definitions

Definition 0: The *label* of a goto is a unique identifier used to specify the target of a goto statement. According to the semantics of COBOL, this label must be a paragraph or section name.

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     GO TO PARA-1.
03
04 PARA-1.
05   MULTIPLY 2 BY VAL1.
```

In the above COBOL code, the goto statement on line `02` has the target labeled `PARA-1` located on line `04`. This means when the goto statement executes, control will pass to the beginning of `PARA-1` and start executing the statement at line `05`.

Definition 1: The *level* of a goto or label is m if the goto or label is nested inside exactly m `loop`, `switch`, or `if/else` statements.

In the above COBOL snippet, the goto statement on line `02` has level = 1 because it is nested inside one `IF` statement. The label `PARA-1` has level = 0. Labels in COBOL (i.e. paragraph or section names) will *always* have level = 0 because they cannot be directly nested inside other statements.

Definition 2: A goto and label are *siblings* if there exists some statement sequence, `stmt_1; ... ; stmt_n`, such that the label statement corresponds to some `stmt_i` and the goto statement corresponds to some `stmt_j` in the statement sequence. For COBOL it is equivalent to say a goto statement g and label l are *siblings* if and only if

$$level(g) = level(l) = 0$$

The following code snippets show examples of goto statements and labels that are siblings.

```
00 PROCEDURE DIVISION.
01  ADD 1 TO VAL1.
02  IF VAL1 IS EQUAL TO 9
03    DIVIDE VAL1 BY 3 GIVING VAL1.
04  GO TO PARA-1.
05  SUBTRACT 3 FROM VAL1.
06
07  PARA-1.
08    MULTIPLY 2 BY VAL1.
```

(`04 GO TO PARA-1` and `07 PARA-1` are siblings)

```
00 PROCEDURE DIVISION.
01  ADD 1 TO VAL1.
02  IF VAL1 IS EQUAL TO 9
03    DIVIDE VAL1 BY 3 GIVING VAL1.
04  GO TO PARA-2.
05  SUBTRACT 3 FROM VAL1.
06
07  PARA-1.
08    MULTIPLY 2 BY VAL1.
09
10  PARA-2.
11    ADD 3 TO VAL1.
```

(`04 GO TO PARA-2` and `10 PARA-2` are siblings)

```
00 PROCEDURE DIVISION.
01  ADD 1 TO VAL1.
02  IF VAL1 IS EQUAL TO 9
03    DIVIDE VAL1 BY 3 GIVING VAL1.
05  SUBTRACT 3 FROM VAL1.
06
07  PARA-1.
08    MULTIPLY 2 BY VAL1.
09
10  PARA-2.
```

```
11    ADD 3 TO VAL1.
12    GO TO PARA-1.
```

(`12 GO TO PARA-1` and `07 PARA-1` are siblings)

The following code snippets show examples of goto statements and labels that are *not* siblings.

```
00 PROCEDURE DIVISION.
01  ADD 1 TO VAL1.
02  IF VAL1 IS EQUAL TO 9
03    GO TO PARA-1.
04  SUBTRACT 3 FROM VAL1.
05
06  PARA-1.
07    MULTIPLY 2 BY VAL1.
```

(`03 GO TO PARA-1` and `06 PARA-1` are *not* siblings because `03 GO TO PARA-1` has level = 1 as a result of being nested inside `02 IF VAL1 IS EQUAL TO 9`)

```
00 PROCEDURE DIVISION.
01  ADD 1 TO VAL1.
02  IF VAL1 IS EQUAL TO 9
03    DIVIDE VAL1 BY 3 GIVING VAL1.
05  SUBTRACT 3 FROM VAL1.
06
07  PARA-1.
08    MULTIPLY 2 BY VAL1.
09
10  PARA-2.
11    ADD 3 TO VAL1.
12    IF VAL1 IS LESS THAN 99
13      GO TO PARA-1.
```

(`13 GO TO PARA-1` and `07 PARA-1` are *not* siblings because `13 GO TO PARA-1` has level = 1 as a result of being nested inside `12 IF VAL1 IS LESS THAN 9`)

Definition 3: A label statement and a goto statement are *directly-related* if there exists some statement sequence, `stmt_1; ... ; stmt_n`, such that either the label or goto statements corresponds to some `stmt_i` and the matching goto or label statement is nested inside some `stmt_j` in the statement sequence.

Because of the semantics of COBOL, a goto statement and a label will always be either *siblings* or *directly-related*. This is because it is impossible to nest a label (i.e. a paragraph or section name) inside another statement (such as an `IF`). Therefore, even though the goto and label are *not* siblings in the previous two code examples above, they *are* directly-related.

The *TCF* paper includes two other definitions—*offset* and *indirectly-related*—but neither is relevant to the semantics and structure of COBOL.

1.1.1.1 Summary of Relevant Points to COBOL

Although discussed above, it is useful to reiterate what parts of the definitions are relevant to COBOL and why. So in summary: 1) Labels in COBOL

(i.e. paragraph or section names) will *always* have level = 0 because they cannot be directly nested inside statements (such as an `IF` statement). 2) Because of (1), a goto statement and a label will always be either *siblings* or *directly-related*. If they are *siblings* then

$$level(goto) = level(label) = 0$$

Otherwise they will be *directly-related* with

$$level(label) = 0 \text{ and } level(goto) > 0$$

Due to these restrictions to how COBOL handles goto statements and labels, only a subset of the transformations found in *TCF* are necessary. A discussion of the COBOL relevant transformations follows.

1.1.2 Transformations

There are two categories of transformations outlined in *TCF*, those that **eliminate the goto statement** and those that **move the goto statement** by unnesting it from other statements in preparation for elimination. We will begin our discussion with the latter.

1.1.2.1 Moving Goto Statements by Unnesting *TCF* defines two types of movement transformations that can be done on a goto statement: 1) **Outward-movement** transformations where a goto or label statement is unnested from and moved outside another statement such as a `loop`, `switch`, or `if/else`. If $level(goto) > level(label)$ then a series of outward-movement transformations are done to decrease the level of the goto statement until $level(goto) = level(label)$ 2) **Inward-movement** transformations where a goto or label statement is nested inside another statement such as a `loop`, `switch`, or `if/else`. If $level(goto) < level(label)$ then a series of inward-movement transformations are done to increase the level of the goto statement until $level(goto) = level(label)$.

Consistent with the discussion on applying the definitions to COBOL in the previous section, inward-movement transformations can be ignored; they are not relevant to COBOL programs. Recall that not only will it *always* be the case for every COBOL program that

$$level(goto) \geqslant level(label),$$

it will also always be true that

$$level(label) = 0.$$

Thus, *the only movement transformation we need in eliminating gotos from a COBOL program is the outward-movement transformation.* Furthermore, the goal of these outward-movement transformations is to make the COBOL goto and target label (the paragraph or section name) *siblings* with

$$level(goto) = level(label) = 0.$$

1.1.2.1.1 Outward-movement Transformations in COBOL There are two basic statements from which gotos can be unnested and moved out to a lower level. These are from inside `IF ... ELSE ...` statements and `PERFORM ... UNTIL ...` statements (the standard looping structure in COBOL).

1.1.2.1.2 Moving Goto Outside an IF Statement The same basic approach is used if the GOTO is nested inside an `IF ... ELSE ...` statement or `PERFORM ... UNTIL ...` statement: - identify the guard expression in the statement in which goto statement is nested, - assign a boolean variable the value of that guard expression, - place this assignment statement right before the nesting statement, and - move the goto statement down one level by pulling it out of the statement it's nested in.

A simple example follows.

```
00 PROCEDURE DIVISION.                  00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9             01   cond_1 = VAL1 IS LESS THAN 9.
02     GO TO PARA-1.                    02   IF VAL1 IS LESS THAN 9
03   COMPUTE VAL1 = VAL1 + 1.    ==>    03     GO TO PARA-1.
04                                       04   COMPUTE VAL1 = VAL1 + 1.
05 PARA-1.                              05
06   MULTIPLY 2 BY VAL1.                06 PARA-1.
                                        07   MULTIPLY 2 BY VAL1.
```

We can see that the `IF` statement on line `01` guards the goto statement on line `02`. The guard expression inside this statement is `VAL1 IS LESS THAN 9`. So we first create a boolean variable at line `01` on the right.

Note that the introduction of this `cond_1` value is not valid COBOL. Here we used a simplified notation, but COBOL does not include boolean valued variables. Level 88 variables are often used to serve this function and the actual implementation of the outward-movement transformation will have to do something similar. See the discussion below in **COBOL Does Not Have Boolean Valued Variables**

Once the value of the guard expression in the conditional in which the goto is nested is captured, we can move the goto statement outside the conditional:

```
00 PROCEDURE DIVISION.                  00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9.      01   cond_1 = VAL1 IS LESS THAN 9.
02   IF VAL1 IS LESS THAN 9             02   IF VAL1 IS LESS THAN 9.
03     GO TO PARA-1.               ==>  03   GO TO PARA-1.
04   COMPUTE VAL1 = VAL1 + 1.           04   COMPUTE VAL1 = VAL1 + 1.
05                                      05
06 PARA-1.                              06 PARA-1.
07   MULTIPLY 2 BY VAL1.                07   MULTIPLY 2 BY VAL1.
```

This concludes the steps involved in the outward-movement transformation of the goto included in this example. How we use the variable `cond_1` that we introduced or what we do with the statement on line `04` after the goto but before the label will be discussed when explaining the goto elimination transformations. Until then, let's look at another example that uses a `PERFORM ... UNTIL ...` to nest a goto.

1.1.2.1.3 Moving Goto Outside a PERFORM Statement Consider the following example.

```
00 PROCEDURE DIVISION.
01   PERFORM UNTIL VAL1 IS GREATER THAN 10
02     GO TO PARA-1.
03   COMPUTE VAL1 = VAL1 + 1.
04
05 PARA-1.
```



```
05   MULTIPLY 2 BY VAL1.

==Transforms to==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN 10.
02   PERFORM UNTIL VAL1 IS GREATER THAN 10.
03   GO TO PARA-1.
04   COMPUTE VAL1 = VAL1 + 1.
05
06 PARA-1.
07   MULTIPLY 2 BY VAL1.
```

Just as in the `IF ... Else ...` statement, we identify the guard expression guarding the goto statement, assign to a boolean variable the value of that guard expression, and move the goto outside the statement in which it was originally nested.

1.1.2.2 Outward-movement Transformations for Multiple Levels of Nesting The above outward-movement process can be done recursively for any level of nesting until $level(goto) = 0$ as can be seen in the following example.

```
00 PROCEDURE DIVISION.                  00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0             01   cond_2 = VAL1 IS LESS THAN 0.
02     SUBTRACT VAL1 FROM VAL1          02   IF VAL1 IS LESS THAN 0
03     IF VAL1 IS EQUAL TO 0            03     SUBTRACT VAL1 TO VAL1
04       MOVE 9 TO VAL1                 04     cond_1 = VAL1 IS EQUAL TO 0
05       GO TO PARA-1            ==>    05     IF VAL1 IS EQUAL TO 0
06     END-IF                           06       MOVE 9 TO VAL1
07   END-IF.                            07     END-IF
08   COMPUTE VAL1 = VAL1 + 1.           04   END-IF.
09                                      05   GO TO PARA-1.
10 PARA-1.                              06   COMPUTE VAL1 = VAL1 + 1.
11   MULTIPLY 2 BY VAL1.                07
                                        10 PARA-1.
                                        11   MULTIPLY 2 BY VAL1.
```

1.1.2.3 Goto Elimination Transformations Once the goto has been completely unnested (i.e. $level(goto) = level(label) = 0$) using the outward-movement transformations we are ready to eliminate the goto statement. There are two such transformations depending on how the goto and its target label are ordered *according to the execution of those statements*. Execution order will become relevant shortly, but let us first describe the two transformations. These transformations are 1) **Forward goto transformation** where a goto that is before its target label according to the execution order of the program is eliminated, and 2) **Backward goto transformation** where a goto that is after its target label according to the execution order of the program is eliminated.

1.1.2.3.1 Forward Goto Transformation The basic procedure of either elimination transformations is to 1) Find the statements that are between the goto and its target label according to the execution order of the program. 2) Nest those statements into the appropriate COBOL control flow statement while preserving equivalence to the original program

Consider the following example where a forward goto is eliminated

```
00 PROCEDURE DIVISION.                00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9           01   cond_1 = VAL1 IS LESS THAN 9.
02     GO TO PARA-1.                  02   IF VAL1 IS LESS THAN 9.
03   COMPUTE VAL1 = VAL1 + 1.   ==>   03   GO TO PARA-1.
04                                     04   COMPUTE VAL1 = VAL1 + 1.
05 PARA-1.                             05
06   MULTIPLY 2 BY VAL1.               06 PARA-1.
                                       07   MUTIPLY 2 BY VAL1.
```

First the goto is moved outward using an outward-movement transformation

```
00 PROCEDURE DIVISION.                   00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9.       01   cond_1 = VAL1 IS LESS THAN 9.
02   IF VAL1 IS LESS THAN 9.             02   IF VAL1 IS LESS THAN 9.
03   GO TO PARA-1.                  ==>  03   IF NOT(cond_1)
04   COMPUTE VAL1 = VAL1 + 1.            04     COMPUTE VAL1 = VAL1 + 1.
05                                       05   GO TO PARA-1.
06 PARA-1.                               06
07   MULTIPLY 2 BY VAL1.                 07 PARA-1.
                                         08   MULTIPLY 2 BY VAL1.
```

Then the goto is eliminated by first collecting the statements between the goto on line `03` and the label `PARA-1` on line `06` on the left. There is only one such statement on line `04`: `COMPUTE VAL1 = VAL1 + 1`. We then create a new `IF` statement with the the negation of the boolean variable `cond_1` on line `01` (created during the outward-movement transformation) as the guard expression. We then nest the collected statements (here only `COMPUTE VAL1 = VAL1 + 1`) inside the true branch of that created `IF` statement. We can now simply remove the goto at line `05` on the right to eliminate it from the program. An optional cleanup step in this particular example is to also removing the `IF` at line `02` on the right since it contains no other statements. As we will see, if it *did* contain other statements this cleanup step would not be possible.

Here we can see the original program on the left and the goto-free, cleaned up version on the right.

```
00 PROCEDURE DIVISION.                00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9           01   cond_1 = VAL1 IS LESS THAN 9.
02     GO TO PARA-1.                  02   IF NOT(cond_1)
03   COMPUTE VAL1 = VAL1 + 1.   ==>   03     COMPUTE VAL1 = VAL1 + 1.
04                                     04
05 PARA-1.                             05 PARA-1.
06   MULTIPLY 2 BY VAL1.               06   MULTIPLY 2 BY VAL1.
```

These two programs are equivalent w.r.t. how the program transforms information.

This procedure of forward goto elimination is the same for any amount of outward-movement transformations

that were required beforehand or any number or type of statements that exist between the goto and its target label according to the execution order of the program:

```
00 PROCEDURE DIVISION.                    00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0               01   cond_2 = VAL1 IS LESS THAN 0.
02     SUBTRACT VAL1 FROM VAL1            02   IF VAL1 IS LESS THAN 0
03     IF VAL1 IS EQUAL TO 0              03     SUBTRACT VAL1 FROM VAL1
04       MOVE 9 TO VAL1                   04     cond_1 = VAL1 IS EQUAL TO 0
05       GO TO PARA-1              ==>    05     IF VAL1 IS EQUAL TO 0
06     END-IF                             06       MOVE 9 TO VAL1
07   END-IF.                              07     END-IF
08   IF VAL1 IS GREATER THAN 8            08   END-IF.
09     ADD VAL1 TO VAL1.                  09   IF NOT(cond_1) OR NOT(cond_2)
10   COMPUTE VAL1 = VAL1 + 1.             10     IF VAL1 IS GREATER THAN 8
11                                        11       ADD VAL1 TO VAL1
12  PARA-1.                               12     END-IF
13    MULTIPLY 2 BY VAL1.                 13     COMPUTE VAL1 = VAL1 + 1
                                          14   END-IF.
                                          15
                                          16 PARA-1.
                                          17   MULTIPLY 2 BY VAL1.
```

If more than one outward-movement transformation was done to completely unnest the goto, the guard expression for the created `IF` statement on line `09` on the right becomes the disjunction of the negation of each of the boolean variables (`cond_1` and `cond_2` in this case).

Note: If the boolean variable was created by unnesting from a `PERFORM ... UNTIL ...` statement, then the guard disjunction will include that variable not* negated*. For example, say we have `cond_1` created by an outward-movement transformation from an `IF` statement and `cond_2` created by an outward-movement transformation from a `PERFORM ... UNTIL ...` statement. Our `IF` statement created during the forward goto elimination transformation will look like this: `IF NOT(cond_1) OR cond_2`. We now return to our original example.

Notice that we cannot cleanup by removing the `IF` statements on lines `02` and `05` on the right in which the goto was originally nested because they included other statement (lines `03` and `06`), the removal of which would change the semantics from the original program.

Furthermore, forward goto elimination must be done recursively, using a combination of outward-movement and forward goto elimination transformations for each level of nesting out of which the goto is pulled. For example,

```
00 PROCEDURE DIVISION.                    00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0               01   IF VAL1 IS LESS THAN 0
02     SUBTRACT VAL1 FROM VAL1            02     SUBTRACT VAL1 TO VAL1
03     IF VAL1 IS EQUAL TO 0              03     cond_1 = VAL1 IS EQUAL TO 0
04       MOVE 9 TO VAL1                   04     IF VAL1 IS EQUAL TO 0
05       GO TO PARA-1                     05       MOVE 9 TO VAL1
06     ELSE                               06     ELSE
```



```
07       MOVE 200 TO VAL1
08     END-IF                    ==>
09     DISPLAY 'Hello, reader'
10   END-IF.
11   IF VAL1 IS GREATER THAN 8
11     ADD VAL1 TO VAL1.
12   COMPUTE VAL1 = VAL1 + 1.
13
14  PARA-1.
15   MULTIPLY 2 BY VAL1.
```

```
07       MOVE 200 TO VAL1
08     END-IF
09     IF NOT(cond_1)
10       DISPLAY 'Hello, reader'
11     END-IF
12     GO TO PARA-1
13   END-IF.
14   IF VAL1 IS GREATER THAN 8
15     ADD VAL1 TO VAL1.
16   COMPUTE VAL1 = VAL1 + 1.
17
18 PARA-1
19   MULTIPLY 2 BY VAL1.
```

We can see in this example that the goto on line `05` on the left is nested inside the true branch of the `IF` statement on line `03`, which is nested inside the true branch of the `IF` statement on line `01`. We can also see that there is a statement at line `09` that ends the true branch of the line `01` statement after the nested conditional statement online `03`. We can see on the right that this statement `DISPLAY 'Hello, reader'` gets nested inside the true branch of the condition guarded by the negation of `cond_1`, which can be seen on line `09` on the right. The goto is now the last statement in the true block of the outer `IF` statement.

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0
02     SUBTRACT VAL1 TO VAL1
03     cond_1 = VAL1 IS EQUAL TO 0
04     IF VAL1 IS EQUAL TO 0
05       MOVE 9 TO VAL1
06     ELSE
07       MOVE 200 TO VAL1
08     END-IF
09     IF NOT(cond_1)                ==>
10       DISPLAY 'Hello, reader'
11     END-IF
12     GO TO PARA-1
13   END-IF.
14   IF VAL1 IS GREATER THAN 8
15     ADD VAL1 TO VAL1.
16   COMPUTE VAL1 = VAL1 + 1.
17
18  PARA-1.
19    MULTIPLY 2 BY VAL1.
```

```
00 PROCEDURE DIVISION.
01   cond_2 = VAL1 IS LESS THAN 0
02   IF VAL1 IS LESS THAN 0
03     SUBTRACT VAL1 TO VAL1
04     cond_1 = VAL1 IS EQUAL TO 0
05     IF VAL1 IS EQUAL TO 0
06       MOVE 9 TO VAL1
07     ELSE
08       MOVE 200 TO VAL1
09     END-IF
10     IF NOT(cond_1)
11       DISPLAY 'Hello, reader'
12     END-IF
13   END-IF.
14   IF NOT(cond_1) OR NOT(cond_2)
15     IF VAL1 IS GREATER THAN 8
16       ADD VAL1 TO VAL1
17     END-IF
18     COMPUTE VAL1 = VAL1 + 1
19   END-IF.
20   GO TO PARA-1.
21
22 PARA-1.
23   MULTIPLY 2 BY VAL1.
```

Once again, the outward-movement transformation is applied to the goto statement left line `12`, the `IF`

statement is created on right line 14 with the negated disjunction of created boolean variables as the guard expression and the collected statements nested inside the true branch, and the goto statement is moved right next to its target label. We can now safely eliminate the goto statement.

```
00 PROCEDURE DIVISION.                  00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0             01   cond_2 = VAL1 IS LESS THAN 0
02     SUBTRACT VAL1 FROM VAL1          02   IF VAL1 IS LESS THAN 0
03     IF VAL1 IS EQUAL TO 0            03     SUBTRACT VAL1 TO VAL1
04       MOVE 9 TO VAL1                 04     cond_1 = VAL1 IS EQUAL TO 0
05       GO TO PARA-1                   05     IF VAL1 IS EQUAL TO 0
06    ELSE                              06       MOVE 9 TO VAL1
07       MOVE 200 TO VAL1               07     ELSE
08    END-IF                            08       MOVE 200 TO VAL1
09    DISPLAY 'Hello, reader'    ==>    09     END-IF
10   END-IF.                            10     IF NOT(cond_1)
11   IF VAL1 IS GREATER THAN 8          11       DISPLAY 'Hello, reader'
11     ADD VAL1 TO VAL1.                12     END-IF
12   COMPUTE VAL1 = VAL1 + 1.           13   END-IF.
13                                      14   IF NOT(cond_1) OR NOT(cond_2)
14  PARA-1.                             15     IF VAL1 IS GREATER THAN 8
15   MULTIPLY 2 BY VAL1.                16       ADD VAL1 TO VAL1
                                        17     END-IF
                                        18     COMPUTE VAL1 = VAL1 + 1
                                        19   END-IF.
                                        20
                                        21 PARA-1.
                                        22   MULTIPLY 2 BY VAL1.
```

Here we can see original program on the left with the fully transformed version on the right with the goto eliminated.

1.1.2.3.2 Backward Goto Transformation To reiterate, the basic procedure of either elimination transformations is to 1) Find the statements that are between the goto and its target label according to the execution order of the program. 2) Nest those statements into the appropriate COBOL control flow statement while preserving equivalence to the original program.

In the forward goto elimination transformation, the appropriate COBOL control flow statement was an `IF` statement. When we are dealing with a backward goto, the appropriate COBOL control flow statement is a `PERFORM .. . UNTIL ... WITH TEST AFTER`. This is COBOL's `do ... while ...` implementation. However, the same behavior can be implemented with a regular `PERFORM ... UNTIL` by duplicating the body of the loop once before entering the loop itself. Here is a small example

```
00 PROCEDURE DIVISION.              00 PROCEDURE DIVISION.
01   PERFORM PARA-1.                01   PERFORM PARA-1.
02   STOP RUN.                      02   STOP RUN.
03                           ==>.   03
04 PARA-1.                          04 PARA-1.
```

```
05   MULTIPLY 2 BY VAL1.
06   IF VAL1 IS LESS THAN 9
07     GO TO PARA-1.
```

```
05   MULTIPLY 2 BY VAL1.
06   cond_1 = VAL1 IS LESS THAN 9.
07   IF VAL1 IS LESS THAN 9.
08   GO TO PARA-1.
```

We can see on right lines 06 through 08, the goto has been unnested from the conditional statement using an outward-movement transformation.

```
00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   MULTIPLY 2 BY VAL1.
06   cond_1 = VAL1 IS LESS THAN 9.
07   IF VAL1 IS LESS THAN 9.
08   GO TO PARA-1.

==Transforms to==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   GO TO PARA-1.
06   PERFORM UNTIL NOT(cond_1) WITH TEST AFTER
07     MULTIPLY 2 BY VAL1
08     cond_1 = VAL1 IS LESS THAN 9
09     IF VAL1 IS LESS THAN 9
```

Then the top statements on lines 05 through 07 are collected and nested inside the body of the `PERFORM UNTIL NOT(cond_1) WITH TEST AFTER` where `cond_1` is the boolean variable created during the outward-movement transformation.

The goto statement can now be eliminated and the conditional statement on bottom line 09 can safely be removed (again, because both the true block and false block of that conditional statement are now empty). The original program with the final eliminated goto version can be seen below.

```
00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   MULTIPLY 2 BY VAL1.
06   IF VAL1 IS LESS THAN 9.
07     GO TO PARA-1.
```

```
==Transforms to==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   PERFORM UNTIL NOT(cond_1) WITH TEST AFTER
06     MULTIPLY 2 BY VAL1
07     cond_1 = VAL1 IS LESS THAN 9
```

Just as in the forward goto elimination transformation, multiple outward-movement transformations can be required to unnest the goto from several statements before completing the backward goto elimination transformation:

```
00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
06   PERFORM UNTIL VAL1 IS GREATER THAN 20
07     MULTIPLY 2 BY VAL1
08     IF VAL1 IS LESS THAN 10
09       MOVE 10 TO VAL1
10       GO TO PARA-1
11     END-IF
12   END-PERFORM.

==Transforms to==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   PERFORM UNTIL NOT(cond_1) OR cond_2 WITH TEST AFTER
06     cond_2 = VAL1 IS GREATER THAN 20
07     PERFORM UNTIL VAL1 IS GREATER THAN 20
08       MULTIPLY 2 BY VAL1
09       cond_1 = VAL1 IS LESS THAN 10
10       IF VAL1 IS LESS THAN 10
11         MOVE 10 TO VAL1
12       END-IF
14     END-PERFORM
15   END-PERFORM.
```

Again, note that `cond_1` is negated in our created disjunction guard in the statement on line `05` below and

cond_2 is not negated. This is because cond_1 was created by moving the goto out of an IF statement and cond_2 was created by moving the goto out of a PERFORM ... UNTIL ... statement.

1.2 Other Considerations When Applying *TCF* to COBOL Source

We have now completed the basic application of the *TCF* paper to COBOL. What follows is a discussion of specific issues that must be considered when applying *TCF* to our work at Phase Change. This includes specific design and implementation options and choices that must be made to easily transform the COBOL source code into a FOM representation.

1.2.1 Backward Goto Elimination Transformation Using Regular Loops and Statement Duplication.

In the discussion of eliminating backward gotos, we used COBOL do-while construct, the PERFORM ... UNITL ... WITH TEST AFTER. While the semantics are slightly different from a usual do-while, the PERFORM ... UNTIL .. WITH TEST AFTER has the key feature of testing the guard expression after iterating through the loop body at least once. Unfortunately, FOM does not have a do-while statement. Thus, to best prepare the original COBOL program for FOM translation, we can use a regular PERFORM ... UNTIL ... loop and strategically duplicate statements from the created loop body. Consider our example from above:

```
00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   MULTIPLY 2 BY VAL1.
06   IF VAL1 IS LESS THAN 9.
07     GO TO PARA-1.

==Transformation using do-while==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   PERFORM UNTIL NOT(cond_1) WITH TEST AFTER
06     MULTIPLY 2 BY VAL1
07     cond_1 = VAL1 IS LESS THAN 9

==Transformation using regular loop==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
```

```
05   MULTIPLY 2 BY VAL1.
06   cond_1 = VAL1 IS LESS THAN 9.
07   PERFORM UNTIL NOT(cond_1)
08     MULTIPLY 2 BY VAL1
09     cond_1 = VAL1 IS LESS THAN 9
```

In the bottom transformations we can see the duplicated loop body (lines `06` and `07` in the middle transformation) that we have added on bottom lines `05` and `06`. We can then drop the `WITH TEST AFTER` clause from our `PERFORM ... UNTIL ...` statement.

Note: we are not suggesting that the implementation of this transformation produce the do-while version first before producing the regular loop version. The point is, however it is done, a regular loop version can be produced that captures the same behavior. Both the do-while and regular loop transformations are included in these examples for understanding purposes only.

The same can be done for any level of nesting/complexity that exists in the program. For example,

```
00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
06   PERFORM UNTIL VAL1 IS GREATER THAN 20
07     MULTIPLY 2 BY VAL1
08     IF VAL1 IS LESS THAN 10
09       MOVE 10 TO VAL1
10       GO TO PARA-1
11     END-IF
12   END-PERFORM.

==Transformation using do-while==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
05   PERFORM UNTIL NOT(cond_1) OR cond_2 WITH TEST AFTER
06     cond_2 = VAL1 IS GREATER THAN 20
07     PERFORM UNTIL VAL1 IS GREATER THAN 20
08       MULTIPLY 2 BY VAL1
09       cond_1 = VAL1 IS LESS THAN 10
10       IF VAL1 IS LESS THAN 10
11         MOVE 10 TO VAL1
12       END-IF
14     END-PERFORM
15   END-PERFORM.
```

```
==Transformation using regular loop==>

00 PROCEDURE DIVISION.
01   PERFORM PARA-1.
02   STOP RUN.
03
04 PARA-1.
06   cond_2 = VAL1 IS GREATER THAN 20
07   PERFORM UNTIL VAL1 IS GREATER THAN 20
08     MULTIPLY 2 BY VAL1
09     cond_1 = VAL1 IS LESS THAN 10
10     IF VAL1 IS LESS THAN 10
11       MOVE 10 TO VAL1
12     END-IF
14   END-PERFORM.
15   PERFORM UNTIL NOT(cond_1) OR cond_2
16     cond_2 = VAL1 IS GREATER THAN 20
17     PERFORM UNTIL VAL1 IS GREATER THAN 20
18       MULTIPLY 2 BY VAL1
19       cond_1 = VAL1 IS LESS THAN 10
20       IF VAL1 IS LESS THAN 10
21         MOVE 10 TO VAL1
22       END-IF
24     END-PERFORM
25   END-PERFORM.
```

1.2.2 COBOL Does Not Have Boolean Valued Variables

- Using level 88 variables to capture the values of guard expressions guarding gotos during the outward-movement transformations.

1.2.3 Paragraphs and Sections are the Target Labels of Goto Statements.

In *TCF*, the goto statement and its target label are structured in such a way that scope is easily identified. For example, nowhere in *TCF* is there an example where a goto statement passes control to a label of a statement that is inside the scope of a different function than the goto statement. COBOL does indeed have a single scope (i.e., all variables are defined globally), but the semantics of paragraphs and the statements they contain can cause tricky situations during goto eliminations. The three issues are 1) How to properly "nest" the statements contained in a paragraph or section when doing a forward or backward goto elimination transformation, 2) The source line ordering of paragraphs can be different than their execution order, and 3) Fall through edges can cause implicit backward gotos.

We will start our discussion with number one.

1.2.3.1 Properly "Nesting" Paragraph or Section Statements

Consider the following example.

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07
08 PARA-1.
09   COMPUTE VAL1 = VAL1 + 9.
10
11 PARA-2.
12   COMPUTE VAL1 = VAL1 + 1.
```

We can see that we have a goto at line `02` that goes to `PARA-2`. So if we take the true branch of the conditional on line `01`, we execute the statement on line `12` and end execution of the program. If we take the false branch of line `2` we perform `PARA-1` by executing line `09`, return to line `05` and then execute lines `06` through `12`, at which point the program terminates.

So let us now start eliminating the goto on line `02`. We first do an outward-movement transformation:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07
08 PARA-1.
09   COMPUTE VAL1 = VAL1 + 9.
10
11 PARA-2.
12   COMPUTE VAL1 = VAL1 + 1.

==Outward-movement transformation==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   GO TO PARA-2
07   COMPUTE VAL1 = VAL1 * 2.
08
09 PARA-1.
```

```
10   COMPUTE VAL1 = VAL1 + 9.
11
12 PARA-2.
13   COMPUTE VAL1 = VAL1 + 1.
```

Here we can see the creation of our boolean valued variable `cond_1` on line `01` below. The goto is now fully unnested it's level is now 0, same as it's target label of `PARA-2`. So we can now start applying the forward goto elimination transformation to the program:

```
00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   GO TO PARA-2
07   COMPUTE VAL1 = VAL1 * 2.
08
09 PARA-1.
10   COMPUTE VAL1 = VAL1 + 9.
11
12 PARA-2.
13   COMPUTE VAL1 = VAL1 + 1.

==Start of forward goto elimination transformation==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08   END-IF.
09   GO TO PARA-2
10
11 PARA-1.
12   COMPUTE VAL1 = VAL1 + 9.
13
14 PARA-2.
15   COMPUTE VAL1 = VAL1 + 1.
```

In the transformation on the bottom, we can see we have created the necessary conditional statement at line `07` and nested the statement `COMPUTE VAL1 = VAL1 * 2` inside its true branch. The goto now sits right

above PARA-1. We are not done, however, because the statement inside PARA-1 needs to also be nested inside the true branch of this condition according to the original execution order of the statements. How this nesting is achieved needs careful consideration since line 12 is actually inside PARA-1. For example, we can not blindly cut and past line 12 (or any other statements that could be in PARA-1) into to true branch of the created conditional at line 06 because line 04 also executes the statements in PARA-1. There are at least two techniques to correctly structure this program: 1) Duplicate the statements in PARA-1 and nest them correctly while mapping them back to the original source lines, or 2) Be clever about how we use new PERFORM statements.

The first option will work, but could potentially create a large intermediate representation making it difficult to look and and debug issues. The second is potentially much cleaner, but might actually be infeasible in the limit of particularly complex programs. This feasibility/infeasibility question is still open.

1.2.3.1.1 The Duplication Technique We will show what the above example would look like using the duplication method before showing the clever method.

```
00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08   END-IF.
09   GO TO PARA-2
10
11 PARA-1.
12   COMPUTE VAL1 = VAL1 + 9.
13
14 PARA-2.
15   COMPUTE VAL1 = VAL1 + 1.

==Transformed using duplication technique==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08     COMPUTE VAL1 = VAL1 + 9.
09   END-IF.
10   COMPUTE CALC1 = CALC1 + 1.
```



```
11   STOP RUN.
12
13 PARA-1.
14   COMPUTE VAL1 = VAL1 + 9.
15
16 PARA-2.
17   COMPUTE VAL1 = VAL1 + 1.
```

Notice here that we also must duplicate the statements in `PARA-2` at the end of the implicit starting paragraph and a `STOP RUN` at line `11` to ensure we don't fall through and incorrectly execute `PARA-1` and `PARA-2` again. This would break equivalence with the original program. Here we can see the original program on the top and the fully transformed version using duplication:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07
08 PARA-1.
09   COMPUTE VAL1 = VAL1 + 9.
10
11 PARA-2.
12   COMPUTE VAL1 = VAL1 + 1.

==Fully transformed==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08     COMPUTE VAL1 = VAL1 + 9.
09   END-IF.
10   COMPUTE CALC1 = CALC1 + 1.
11   STOP RUN.
12
13 PARA-1.
14   COMPUTE VAL1 = VAL1 + 9.
15
16 PARA-2.
```

```
17   COMPUTE VAL1 = VAL1 + 1.
```

Again, we know this will work, but it might cause a substantial increase in the representation in many cases.

1.2.3.1.2 The Clever Technique The clever technique figures out which paragraphs need to be performed and inserts the appropriate `PERFORM` statements. Here's the example starting at the point at which we must nest `PARA-1` statements inside the created conditional:

```
00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08   END-IF.
09   GO TO PARA-2
10
11 PARA-1.
12   COMPUTE VAL1 = VAL1 + 9.
13
14 PARA-2.
15   COMPUTE VAL1 = VAL1 + 1.

==Transformed using duplication technique==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08     PERFORM PARA-1.
09   END-IF.
10   PERFORM PARA-2.
11   STOP RUN.
12
13 PARA-1.
14   COMPUTE VAL1 = VAL1 + 9.
15
16 PARA-2.
17   COMPUTE VAL1 = VAL1 + 1.
```

Basically what this approach is doing is not allowing any fall through behavior by treating paragraphs as

blocks of code that *must* be called explicitly in the program. Notice that if there were a paragraph between `PARA-1` and `PARA-2` (let us call this `PARA-1-1`), then the inserted `PERFORM` on line `08` would read `PERFORM PARA-1 THRU PARA-1-1`. This is where the uncertainty lies. We are not sure how this option scales with program complexity. It is an open question whether or not the proper reasoning can be done to figure out the correct paragraphs the `PERFORM` needs to execute. It is straightforward in this case, but could become very tricky with a complex combination of `PERFORM`s, gotos, and `STOP RUN`s.

Here is the original program with the transformation using the clever technique:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07
08 PARA-1.
09   COMPUTE VAL1 = VAL1 + 9.
10
11 PARA-2.
12   COMPUTE VAL1 = VAL1 + 1.

==Fully transformed==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS GREATER THAN OR EQUAL TO 99
02   IF VAL1 IS GREATER THAN OR EQUAL TO 99
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   IF NOT(cond_1)
07     COMPUTE VAL1 = VAL1 * 2
08     PERFORM PARA-1.
09   END-IF.
10   PERFORM PARA-2.
11   STOP RUN.
12
13 PARA-1.
14   COMPUTE VAL1 = VAL1 + 9.
15
16 PARA-2.
17   COMPUTE VAL1 = VAL1 + 1.
```

1.2.3.2 Source Line Ordering Differs from Execution Order The other issue with having paragraphs and section names be the labels for goto statements is execution order. In *TCF* a goto with a lower

number than its target label is guaranteed to be a forward goto and vice versa for a goto statement with a higher number than its target label. In COBOL, this is not the case:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07   STOP RUN.
08
09 PARA-1.
10   COMPUTE VAL1 = VAL1 + 9.
11   STOP RUN.
12
13 PARA-2.
14   COMPUTE VAL1 = VAL1 + 1.
15   GO TO PARA-1
```

Notice in this example there are two goto statements: one on line `01` and another on line `15`. Furthermore, from a source line perspective, the goto statement on line `15` is below its target label `PARA-1`. If we inspect the execution order of the program, however, we will see that line `15` is actually a *forward* goto: execution passes to line `14` from line `02` and then to line `10` where the program eventually terminates at line `11`.

1.2.3.3 Fall-throughs Can Cause Implicit Backward Gotos

Related to the above example, consider a slight modification:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS GREATER THAN OR EQUAL TO 99
02     GO TO PARA-2
03   ELSE
04     PERFORM PARA-1
05   END-IF.
06   COMPUTE VAL1 = VAL1 * 2.
07   STOP RUN.
08
09 PARA-1.
10   COMPUTE VAL1 = VAL1 + 9.
11
12 PARA-2.
13   COMPUTE VAL1 = VAL1 + 1.
14   GO TO PARA-1
```

Here we can see that the `STOP RUN` from line `11` is now gone. Line `14` still contains a forward goto, but now we will fall through to `PARA-2` after completing execution of `PARA-1`. This effectively creates an implicit backward goto.

1.2.4 Other Statements Causing Unstructured Exits

The critical issue caused by goto statements is they allow multiple, unstructured exits from the program. To be a fully structured program, all paths through that program must use the same single exit. This raises the question, are there other statements that cause unstructured exits?

1.2.4.1 STOP RUNs

Consider the following example:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     STOP RUN
03   END-IF
04   MUTLIPLY 2 BY VAL1.
05   STOP RUN.
```

Here there are two unstructured exits: one on line `02` and another on line `05`. To make this program structured, we can use all the same definition and transformations discussed above in a novel way to eliminate the exits until we are left with a single structured exit. We first apply the outward-movement transformation:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     STOP RUN
03   END-IF
04   MUTLIPLY 2 BY VAL1.
05   STOP RUN.

==Outward-movement transformation=>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF VAL1 IS LESS THAN 9
03   END-IF
04   STOP RUN
05   MUTLIPLY 2 BY VAL1.
06   STOP RUN.
```

Then we apply the "forward goto" elimination transformation by moving the `STOP RUN` at line `04` next to the `STOP RUN` at line `06`, collecting the statements we move past and nesting them in a created conditional:

```
00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF VAL1 IS LESS THAN 9
03   END-IF
04   STOP RUN
05   MUTLIPLY 2 BY VAL1.
06   STOP RUN.

==Outward-movement transformation=>
```

```
00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF VAL1 IS LESS THAN 9
03   END-IF
04   IF NOT(cond_1)
05     MUTLIPLY 2 BY VAL1.
06   END-IF.
07   STOP RUN
08   STOP RUN.
```

The `STOP RUN` can now be eliminated and the original conditional cleaned up:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     STOP RUN
03   END-IF
04   MUTLIPLY 2 BY VAL1.
05   STOP RUN.

==Full transformation=>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF NOT(cond_1)
03     MUTLIPLY 2 BY VAL1.
04   END-IF.
06   STOP RUN.
```

This works with implicit exits:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     STOP RUN
03   END-IF.
04   MUTLIPLY 2 BY VAL1.

==Transforms to=>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF NOT(cond_1)
03     MUTLIPLY 2 BY VAL1.
04   END-IF.
```

As well as with any level of nesting:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 0
```

```
02     SUBTRACT VAL1 FROM VAL1
03     IF VAL1 IS EQUAL TO 0
04       MOVE 9 TO VAL1
05       STOP RUN
06     END-IF
07   END-IF.
08   IF VAL1 IS GREATER THAN 8
09     ADD VAL1 TO VAL1.
10   COMPUTE VAL1 = VAL1 + 1.

==Transforms to=>

00 PROCEDURE DIVISION.
01   cond_2 = VAL1 IS LESS THAN 0.
02   IF VAL1 IS LESS THAN 0
03     SUBTRACT VAL1 SUBTRACT VAL1
04     cond_1 = VAL1 IS EQUAL TO 0
05     IF VAL1 IS EQUAL TO 0
06       MOVE 9 TO VAL1
07     END-IF
08   END-IF.
09   IF NOT(cond_1) OR NOT(cond_2)
10     IF VAL1 IS GREATER THAN 8
11       ADD VAL1 TO VAL1
12     END-IF
13     COMPUTE VAL1 = VAL1 + 1
14   END-IF.
```

1.2.4.2 PERFORMs that Don't Return THIS SECTION'S EXAMPLES ARE NOT COMPLETE

`PERFORM` statements are a structured way for COBOL programs to execute a well-defined group of statements and cleanly return to the original context in which the `PERFORM` statement was first executed. Unfortunately, when used in combination with `STOP RUN`s and/or gotos, `PERFORM` statements can cause unstructured behavior just as bad as gotos. For example,

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     PERFORM PARA-1
03   END-IF
04   ADD 1 TO VAL1.
05   STOP RUN.
06
07 PARA-1.
07   MUTLIPLY 2 BY VAL1.
08   STOP RUN.
```

Notice there is a `STOP RUN` inside `PARA-1` at line `08`. This causes the conditions under which line `04` gets executed to be ambiguous. The problem is that one branch of the conditional on line `02` (the true branch) never returns to the structured join point after the conditional at line `03`. Luckily, the same transformations used in the goto elimination transformations can be used here to properly structure the program:

```
00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
02     PERFORM PARA-1
03   END-IF
04   ADD 1 TO VAL1.
05   STOP RUN.
06
07 PARA-1.
07   MUTLIPLY 2 BY VAL1.
08   STOP RUN.

==Transforms to==>

00 PROCEDURE DIVISION.
01   IF VAL1 IS LESS THAN 9
07     MUTLIPLY 2 BY VAL1
08     STOP RUN
03   END-IF
04   ADD 1 TO VAL1.
05   STOP RUN.
06
07 PARA-1.
07   MUTLIPLY 2 BY VAL1.
08   STOP RUN.

==Transforms to==>

00 PROCEDURE DIVISION.
01   cond_1 = VAL1 IS LESS THAN 9
02   IF VAL1 IS LESS THAN 9
03     MUTLIPLY 2 BY VAL1
04   END-IF
05   IF NOT(cond_1)
06     ADD 1 TO VAL1.
07   END-IF
08   STOP RUN.
09
10 PARA-1.
11   MUTLIPLY 2 BY VAL1.
12   STOP RUN.
```