goto in Fortran

One does not truly appreciate the long-ranging effects of the **goto** statement until one has encountered an older version of Fortran. All these statements were included in the initial release of Fortran for the IBM 704.

THE UNCONDITIONAL BRANCH

The simplest form of the **go to** statement in Fortran is the unconditional branch. The statement takes the form:

```
GO TO n
```

where **n** is the statement number of another statement in the program. When such a **GO TO** statement is encountered, the next statement executed will be the one having the specified statement number **n**. This simple **GO TO** causes an *unconditional* transfer of control. When it comes to re-engineering unconditional **GO TO** statements - they are often extremely challenging.

THE ARITHMETIC IF

As opposed to the unconditional branch, the arithmetic **IF** provides a mechanism to transfer control if some condition is met. The arithmetic **IF** has the following form:

```
IF (e) n1,n2,n3
```

where **e** stands for any arithmetic expression, and $\mathbf{n_1}$, $\mathbf{n_2}$, and $\mathbf{n_3}$ are statement numbers. If the value of the expression **e** is negative, control is transferred to the statement number $\mathbf{n_1}$; if the value of the expression is zero, control is transferred to $\mathbf{n_2}$; and if the expression is positive, control is transferred to $\mathbf{n_3}$. A simple example of an arithmetic **IF** can be derived from the following formula:

$$Q = \begin{cases} -\frac{\pi}{2} & \text{if } a < 0 \\ 0 & \text{if } a = 0 \\ \frac{\pi}{2} & \text{if } a > 0 \end{cases}$$

This equates to the following program segment:

```
      IF (A) 20, 30, 40
20    Q = -(PI/2.0)
      GO TO 50
30    Q = 0.0
      GO TO 50
```

```
40    Q = PI/2.0
50
```

This **IF** statement involves the examination of the value of the variable A. If the value of the variable is negative, control is transferred to statement 20, which calculates the value of Q. The next statement after this a unconditional jump to statement 50. If the "**GO TO 50**" were not in the code, the program would automatically go on to the next statement in the sequence, which is statement 30. This statement effectively "skips around" the remainder of the code associated with the arithmetic **IF**. The arithmetic **IF** can be re-engineered into a tiered **IF** structure. In the case of the example shown above, the code would become:

```
if (a < 0) then
    q = -(pi/2.0)
else if (a > 0) then
    q = (pi/2.0)
else
    q = 0.0
end if
```

The arithmetic-**if**, can be tricky to remove sometimes. Consider the code below:

```
    do 6 i = 1,3
       if (dif - c(i)) 7,7,6
6   continue
    i = 4
7   ffm = b(i)*exp(a(i)*dif)
```

Here if the value of **dif-c(i)** is ≤0 then the program branches to label 7, otherwise if >0 it branches to label 6, which is the next iteration of the loop. Basically the loop exits when **dif** is less than or equal to **c(i)**, and the value of **i** is used in the calculation of **ffm**. If the condition is never met, then the loop ends, and the value of **i** is set to 4, and **ffm** is calculated.

The first thing to do when re-engineering this, is to update the do-loop to F90+ standards.

```
  do i = 1,3
      if (dif - c(i) .le. 0) then
          go to 7
      end if
  end do
  i = 4
7 ffm = b(i)*exp(a(i)*dif)
```

The problem here is removing the **go to** and still being able to skip the statement **i=4**. The trick is to incorporate **i=4** into the loop. For example:

```
  do i = 1,4
      if (i == 4) then
          exit
      else if (dif <= c(i)) then
          exit
      end if
```

```
    end do
    ffm = b(i)*exp(a(i)*dif)
```

The loop runs through 4 times, when the index **i** becomes 4, the loop exits, and **ffm** is calculated. Otherwise for **i** is 1-3, the loop only exits if **dif** is less than or equal to **c(i)**.

THE COMPUTED GO TO

The arithmetic **IF** provides a Fortran program with a three-way branch. The computed **GO TO** extends this concept by providing an *n*-way branch based on the value of an integer variable. The statement has the general form:

```
GO TO (n1,n1,...,nm), i
```

In this statement, **i** must be an integer variable, and $\mathbf{n_1}$, $\mathbf{n_2}$,..., $\mathbf{n_m}$ are statement numbers of statements elsewhere in the program. If the value of the variable **i** at the time this statement is executed is **j**, then control is transferred to the statement with the statement number $\mathbf{n_j}$. For example, consider the following statement.

```
GO TO (7, 19, 400, 4, 720), ic
```

If the value of the variable **ic** is 1, then control is transferred to statement number 7; if it is 2, to statement 19; if it is 3, statement 400, and so on. The value of the integer variable must be in the range 1 to **m** where **m** denotes how many statement numbers there are in parentheses. Here is an example of a program segment that computes the first five Legendre polynomials:

$$P_0(x) = 1$$
$$P_1(x) = x$$
$$P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$$
$$P_3(x) = \frac{5}{2}x^3 - \frac{3}{2}x$$
$$P_4(x) = \frac{35}{8}x^4 - \frac{15}{4}x^2 + \frac{3}{8}$$

The assumption is that the variable **x** has been calculated previously and a value of between 0 and 4 has been assigned to the variable **LEGNO** which determines which of the Legendre polynomials will be computed. This program segment will calculate this:

```
      LEGEP = LEGNO + 1
      GO TO (20, 30, 40, 50, 60), LEGEP
20    P = 1.0
      GO TO 100
30    P = X
      GO TO 100
40    P = 1.5*X**2 - 0.5
```

```
      GO TO 100
  50  P = 2.5*X**3 - 1.5*X
      GO TO 100
  60  P = 4.375*X**4 - 3.75*X**2 + 0.375
 100
```

The computed **GO TO** is one of the easiest to re-engineer - it can be replaced with a select-case statement. For example the Legendre polynomial example would look like this:

```
select case(legno)
case(0)
    p = 1.0
case(1)
    p = x
case(2)
    p = 1.5*x**2 - 0.5
case(3)
    p = 2.5*x**3 - 1.5*x
case(4)
    p = 4.375*x**4 - 3.75*x**2 + 0.375
case default
    write (*,*) 'Invalid polynomial'
end select
```

THE ASSIGNED GO TO

The assigned GO TO is a variant of the computed GO TO which allows a branch to the statement whose address has previously been assigned to an integer variable. The statement has the general form:

```
GO TO i, (n1,n1,...,nm)
```

where **i** is an integer variable, and $\mathbf{n_1}$, $\mathbf{n_2}$,..., $\mathbf{n_m}$ are statement numbers. Here the value of **i** must be one of the entries in the list.

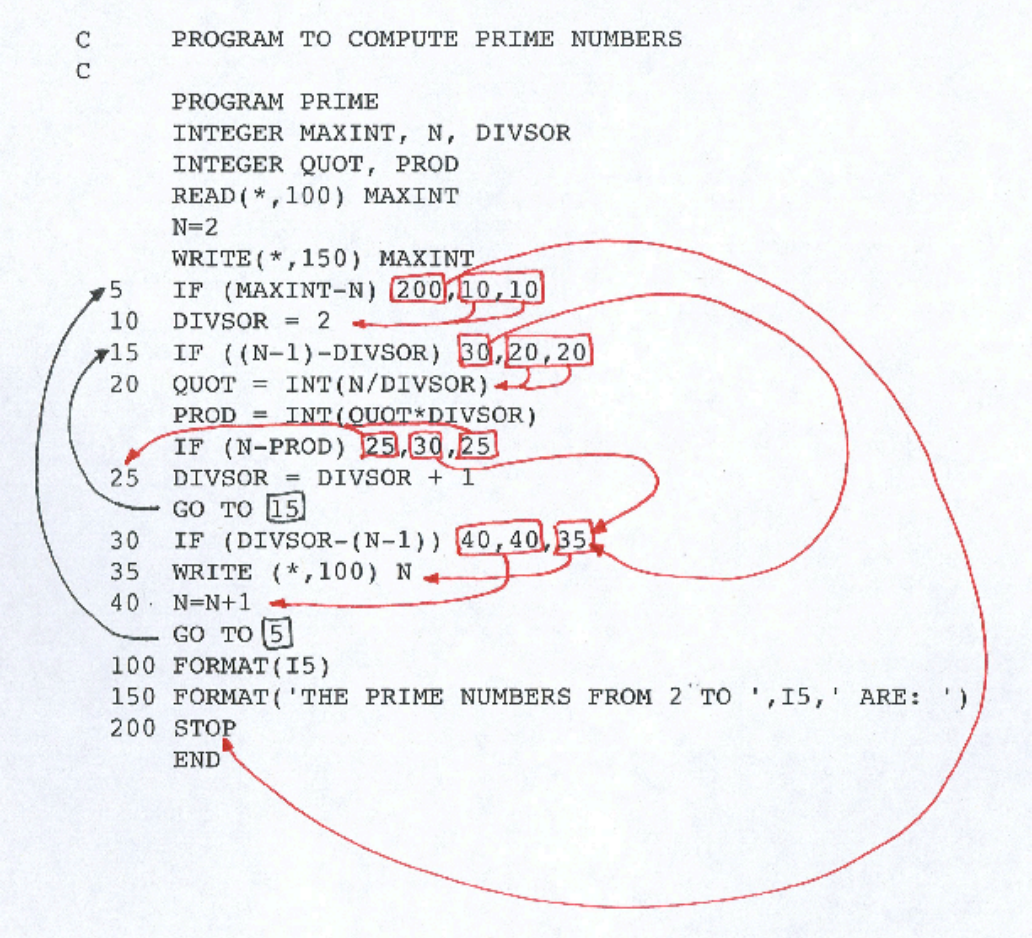
The assigned GO TO is normally used in conjunction with the ASSIGN statement. The ASSIGN statement assigns a statement label value to an integer variable. When this has been done the variable no longer has an arithmetic value. For example:

```
ASSIGN 30 TO i
GO TO i, (10, 20, 30)
```

Consider the following Fortran code containing a series of arithmetic **if** statements, and two **go to** statements which mimic the effect of loops:

A GOOD EXAMPLE

This is an example showing how prolific the **goto** statement was in older Fortran programs. This program is 24 lines in length, yet contains four arithmetic **if** statements, and two unconditional **goto**'s.



```
C       PROGRAM TO COMPUTE PRIME NUMBERS
C
        PROGRAM PRIME
        INTEGER MAXINT, N, DIVSOR
        INTEGER QUOT, PROD
        READ(*,100) MAXINT
        N=2
        WRITE(*,150) MAXINT
 5      IF (MAXINT-N) 200,10,10
 10     DIVSOR = 2
 15     IF ((N-1)-DIVSOR) 30,20,20
 20     QUOT = INT(N/DIVSOR)
        PROD = INT(QUOT*DIVSOR)
        IF (N-PROD) 25,30,25
 25     DIVSOR = DIVSOR + 1
        GO TO 15
 30     IF (DIVSOR-(N-1)) 40,40,35
 35     WRITE (*,100) N
 40     N=N+1
        GO TO 5
 100    FORMAT(I5)
 150    FORMAT('THE PRIME NUMBERS FROM 2 TO ',I5,' ARE: ')
 200    STOP
        END
```