The Leap Year Problem

Arie van Deursen

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

A significant number of programs incorrectly treats the year 2000 as a non-leap year. We list 21 real life code fragments illustrating the large variety of ways that are used to determine whether a given year is a leap year or not. Some of these fragments are correct; others will fail in the year 2000. The fragments are written in C, Pascal, COBOL, and assembly language. We discuss the consequences for automated tool support, as well as the organizational implications of the leap year problem.

1991 Computing Reviews Classification System: D.2.2, D.2.3, D.2.7., D.3.4, F.3.1, I.2.2.

Keywords and Phrases: Software maintenance, program understanding, plan-based concept recovery, COBOL, Year 2000 Problem.

Note: An earlier version appeared in *Dat is dus heel interessant; Liber Amicorum Paul Klint 25 jaar CWI*, pp. 61–73, November 1997, Amsterdam, CWI.

Note: To appear in *The Year/2000 Journal* **2**(4), July/August, 1998.

Note: Work carried out under project SEN-1.1, *Software Renovation*.

Introduction

The Year/2000 problem is about two-digit dates. At least, that is what we read in the newspapers. But there is more to it than two-digit dates. The year 2000 is a leap year. Some programs know about this. They simply check whether a year is divisible by 4, and hence conclude that 2000 is a leap year. Or they are more rigorous and are aware of the exception: a year that is divisible by 100 as well is not a leap year, unless it is also divisible by 400. In other words, neither 1900 nor 2100 are leap years, but 2000 is.

A significant number of programs, however, incorrectly treat the year 2000 as a non-leap year. This may be caused by the use of two digit dates, i.e., "00" is treated as 1900 rather than 2000. In most cases, the programmer simply will have had the wrong algorithm in mind. One common error is the assumption that centuries never are leap years; i.e., the programmer mistakenly forgets the "exception to the exception". The other common error is the idea that the year 2000 simply cannot be a leap year. To paraphrase a recent posting at an Internet newsgroup: *Ok, my info was incorrect - I was taught in school that years divisible by 400 were leap years except for those divisible by 1000 (confused yet — I sure was!)*.

This "leap year problem" is not a toy problem. An example of the cost that might be involved is the $1,000,000 damage caused by the fact that all control computers of a New Zealand aluminum smelter simultaneously went down because they could not deal with the 366th day of 1996 [5]. Similar crashes are to be expected in the year 2000, on a much larger scale.

Unfortunately, it is more difficult to solve this leap year problem than one would expect. In particular, current tool support does not support automated remediation. Therefore, you will have to be fully aware of the nature of this leap year problem, and the implications it has for your organization and your Year/2000 remediation project.

Code Fragments

In the sections to come, we list leap year fragments in Pascal, C, assembly, and COBOL. They come from various sources, such as year 2000 related literature [3, 4], a request posted on `comp.software.year-2000`, and inspections of real life code. (More fragments are still welcome: you can send an email to `arie@cwi.nl`).

Of particular interest are the COBOL fragments, which come from various sources. A large number of fragments is taken from a 50-page list of

date-related code examples as occurring in a set of COBOL programs comprising 10 million lines of code, a list which is discussed in more detail in a separate paper (available via `http://www.cwi.nl/~arie/`) [1].

Moreover, two systems of 100,000 lines were studied in their entirety. It turned out that in each of these systems, four different procedures for checking leap years were implemented, some correct, some incorrect. These figures are a reason for concern. Not only do they illustrate a disheartening lack of code reuse; extrapolating them suggests that a large organization with say a 100 million lines of code (which is not even very large) might maintain 4000 leap year procedures! This is probably too high an estimate, but it would be interesting to learn what the true figure is.

C and Pascal

In most of the modern programming languages, such as Pascal, C, Ada, or Java, it is possible to define a function taking a year as an argument, and returning a Boolean (true or false) value indicating whether the argument given is a leap year or not. COBOL is different in this respect, in that it does not support parameter passing at the section level, nor functions returning a value.

Since leap year code is most easily understood as a function, we first look at some C and Pascal fragments. The fragment below is the typical way of determining leap years in C. It relies on the C `%` operator to determine the remainder of a division, and uses operators such as `&&` (logical and), `||` (logical or), and `!` (negation) to express the conditions for determining leap years concisely.

Fragment 1

```
int isleap(year) int year;
{
return( (year%4   == 0 &&
         year%100 != 0) ||
        year%400 == 0 )
}
```

In other words, a leap year is either divisible by 4 yet not by 100, or it is divisible by 400. The function above returns a Boolean result; the macro below returns the number of days in a year. It uses the C `?` S_1 `:` S_2 if-then-else construct.

Fragment 2

```
#define dysize(A)       \
 ((((A) % 4 == 0 &&     \
    (A) % 100 != 0 ) || \
   (A) % 400 == 0)      \
  ? 366:365 )
```

The logic used to determine leap years can of course be formulated in many different ways. The fragment below uses an outer level logical `&&` (and), rather than an `||` (or), in combination with one extra negation. Here the Boolean result is assigned to a variable called `leap_year`.

Fragment 3

```
leap_year =
  (year % 4 == 0 &&
   !( year % 100 == 0 &&
      !(year % 400 == 0)));
```

The following fragment, taken from the Linux `cal` application, defines a macro that even takes the calendar reorganization of 1752 into account. Years before 1752 are considered leap if they are divisible by 4, independent of divisibility by 100. To determine divisibility by 0, it uses the fact that in C the integer value 0 is used to represent "false", whereas any other integer value means "true". Thus, `!(yr%4)` checks whether `yr` is divisible by 4.

Fragment 4

```
#define leap_year(yr)    \
 ( (yr) <= 1752 ?        \
     !((yr) % 4)         \
   : ( !((yr) % 4) &&    \
       ((yr) % 100) ) || \
     !((yr) % 400) )
```

Evidently, there are many different ways to combine the various divisions into the correct logical expression. As an example, J. Stockton compiled a list of 10 fragments, all in Pascal, illustrating the various ways of combining the logical operators in leap year computations. The original version is available through `http://www.merlyn.demon.co.uk/programs/leapyear.pas`. It includes versions using only `xor`, `<>`, `=`, `and not`, etc. A particularly nice one, using only the "exclusive or" operation, is the following:

Fragment 5

```
Leap :=
  (Y mod 4 = 0)   xor
  (Y mod 100 = 0) xor
  (Y mod 400 = 0) ;
```

The fragments shown so far all compute one Boolean expression for determining leap years. The alternative is to indicate explicitly in which order

the various remainder computations are to be performed. This results in nested if-then-else statements, which are used to set a Boolean flag, as shown below:

Fragment 6

```
if Y mod 400 = 0  then
  Leap := true
else if Y mod 100 = 0 then
    Leap := false
  else if Y mod 4 = 0 then
      Leap := true
    else Leap := false ;
```

The C `return` statement introduces further possibilities. The `return` statement jumps out of a procedure or function call, returning a certain value to the calling procedure.

Fragment 7

```
typedef
  unsigned char Boolean;
enum _boolEnum {
  false=0,
  true=1
};

Boolean IsLeapYear(int y)
{ if (y % 400 == 0)
    return true;
  if (y % 100 == 0)
    return false;
  if (y % 4 == 0)
    return true;
  return false;
}
```

Note that in this case the order of the statements is absolutely crucial: the division by 400 should be made first, and only if that fails the division by 100 is certain to yield a non-leap year.

Correct COBOL

In COBOL, it is not so simple to define a function returning a Boolean result. Therefore, most leap year fragments use explicit IF-THEN statements to guide which remainders are to be computed, followed by either the leap year related code or a move of a flag into a certain variable, as shown in the fragment below. The `PIC` clauses of that fragment are variable declarations. They define a `DATE` record, consisting of three fields, `DAY`, `MONTH` (both two digits wide, indicated by `99`), and `YEAR`, which consumes four digits. This `YEAR` field is subdivided in two `CC` century and two non-century `YY` digits.

Fragment 8

```
01 DATE.
   02 DAY       PIC   99.
   02 MONTH     PIC   99.
   02 YEAR      PIC   9999
   02 CCYY REDEFINES YEAR.
      03 CC     PIC   99.
      03 YY     PIC   99.
01 LEAP         PIC   X.

MOVE 'F' TO LEAP.
DIVIDE YEAR BY 4
  GIVING Q REMAINDER R-1.
IF R-1 = 0
  IF YY = 0
    DIVIDE YEAR BY 400
      GIVING Q REMAINDER R-2
    IF R-2 = 0
      MOVE 'T' TO LEAP
    END-IF.
  ELSE
    MOVE 'T' TO LEAP
  END-IF
END-IF
```

Observe that this fragment does not contain a division of the year by 100. Instead, it checks whether the `YY` part of the year equals zero. If that is the case, the full `YEAR` must represent a century.

The fragment below executes essentially the same statements. However, the order is completely different, making it not obvious to see that it is indeed the same. Also, the `YEAR` is not divided by 400; instead just the `CC` part is divided by 4, with an equivalent result.

Fragment 9

```
IF YY EQUAL ZERO
  DIVIDE CC BY 4
    GIVING Q REMAINDER R
ELSE
  DIVIDE YY BY 4
    GIVING Q REMAINDER R
END-IF
IF R EQUAL ZERO
  MOVE 29 TO MONTH-DAYS(2)
ELSE
  MOVE 28 TO MONTH-DAYS(2)
END-IF
```

Significantly more complex are procedures that use GOTOs to implement the leap year logic. The

following fragments computes all three divisions, by 4, 100, and 400, and uses GOTOs to determine the order in which this is to be done. The fragment is correct, but finding that out may take some time.

Fragment 10

```
  MOVE 28 TO DM(2).
  DIVIDE YEAR BY 4
    GIVING Q REMAINDER R-1.
  IF R-1 NOT EQUAL ZERO
    GO TO L-100
  END-IF.
  DIVIDE YEAR BY 400
    GIVING Q REMAINDER R-1.
  DIVIDE YEAR BY 100
    GIVING Q REMAINDER R-2.
  IF R-1 NOT EQUAL ZERO AND
     R-2 EQUAL ZERO
    GO TO L-100
  END-IF.
  MOVE 29 TO DM(2).
L-100.
  [ ... ]
```

Division by 4

All examples we have seen so far contain the robust, full leap year determination check, including the divisions by 100 and 400. From the year 2000 perspective, however, simply dividing by 4 is generally considered sufficiently safe as well. The first error occurring due to this simpler procedure will be in 2100, a problem we can safely postpone to the next millennium. Therefore, the simple fragment shown below can be considered correct:

Fragment 11

```
DIVIDE YEAR BY 4
  GIVING Q REMAINDER R.
IF R = 0
  [ leap year code ]
END-IF
```

Even such a simple division by 4 can be implemented in many different ways. One alternative is to use a COMPUTE statement in the following way:

Fragment 12

```
COMPUTE Q = YEAR / 4.
COMPUTE R = YEAR - (Q * 4).
```

Much trickier is the next way of computing the remainder in COBOL. Here a division by 4 is computed, and the result is stored in a variable `TMP` which is declared in such a way that it can only hold two digits *behind* the decimal point, as defined by the `PIC V9(02)` code. In other words, this variable will hold the values .00, .25, .50, or .75. If it is zero, the year was divisible by 4.

Fragment 13

```
  05 TMP   PIC V9(02).

COMPUTE TMP = YY / 4.
IF TMP EQUAL ZERO
  MOVE 29 TO DD(2)
ELSE
  MOVE 28 TO DD(2)
END-IF.
```

Things can be further complicated if multiple functionality is mixed into one statement. The following example first simply divides the year by 4. It then uses the result to determine whether not only the current year is a leap year, but also whether last year was one.

Fragment 14

```
DIVIDE YEAR BY 4
  GIVING Q REMAINDER R.
IF R EQUAL 0
  MOVE 'T' TO
    LEAP-THIS-YEAR
ELSE
  IF R EQUAL 1
    MOVE 'T' TO
      LEAP-LAST-YEAR
  END-IF
END-IF
```

Of particular interest is the fragment below. It uses variables `R8` and `R9` to determine the divisibility of the variable `YYB` by 4. As suggested by these variable names this fragment was originally an assembly program. It was automatically re-engineered to COBOL, but unnecessary register assignments were not eliminated in this process.

Fragment 15

```
COMPUTE YYT = (YYB - 1)*365
MOVE YYB TO R9 R8
COMPUTE R9 = R9 / 4
COMPUTE R8 = R8 - 4 * R9
IF R8 = 0 AND MMB <= 2
  SUBTRACT 1 FROM R9
END-IF
MOVE R9 TO STT
```

Note that it is not the year variable itself, YYB, which is divided by 4, but a variable into which the year has been moved. Some form of dataflow analysis will be required to detect such dependencies.

The assembly code from which this COBOL fragment was derived is shown in Figure 1. It is part of a program for computing the days of the week based on an "eternal" calendar which runs from 1901 until 1999. The program itself will be incorrect from 2000 onwards (it will consider 02 to be 1902 rather 2002). The leap year computation fragment, however, does not take century years into account, and therefore happens to behave correctly from 1901 until 2099.

Incorrect COBOL

The fragments shown so far use a wide variety of language constructs, yet they all provide a correct implementation of a leap year check. The fragments still to come, however, are incorrect.

Surprisingly many leap year procedures found in real life code contain the error to make all centuries non-leap years:

Fragment 17

```
DIVIDE YEAR BY 4
  GIVING Q REMAINDER R-1.
DIVIDE YEAR BY 100
  GIVING Q REMAINDER R-2.
IF R-1 = 0 AND
   R-2 NOT = 0
THEN
  [ leap year code ]
END-IF
```

The same error of considering all centuries as non-leap years is contained in the code below. The order of statements, though, is again completely different.

Fragment 18

```
IF YY EQUAL ZERO
  MOVE 28 TO MONTH-DAYS(2)
ELSE
  DIVIDE YEAR BY 4
    GIVING Q REMAINDER R
  IF R EQUAL ZERO
    MOVE 29 TO MONTH-DAYS(2)
  ELSE
    MOVE 28 TO MONTH-DAYS(2)
  END-IF
END-IF
```

The next example can be viewed as an attempt to implement a function returning a result in COBOL. The actual leap check is contained in a separate section. The result is stored in the variable TMP. This variable makes use of a special 88 predicate field to realize Boolean values: it is true only if the TMP variable equals zero.

Unfortunately, the code in the CHECK-LEAP section is incorrect, explicitly moving 1 into TMP if the year is a century year.

Fragment 19

```
01 TMP          PIC 99.
  88 LEAP-YEAR  VALUE ZERO.

   PERFORM CHECK-LEAP
   IF LEAP-YEAR
     ADD 186 TO DC
   ELSE ADD 185 TO DC
   END-IF.

CHECK-LEAP.
   IF YY = ZERO
     MOVE 1 TO TMP
   ELSE
     DIVIDE YEAR BY 4
       GIVING Q
         REMAINDER TMP
   END-IF.
```

The DC variable is probably a day counter, which was earlier filled with 180 (approximately half a year).

In addition to algorithms refusing to consider century years as leap years, there are leap year procedures that explicitly check that the year in question is not the year 2000:

Fragment 20

```
DIVIDE YEAR BY 4
  GIVING Q REMAINDER R.
IF R = 0 AND
   YEAR NOT = 2000
THEN
  [ leap year code ]
END-IF
```

The last fragment we show — real life code! — is also the shortest. It is the ultimate short-term "solution":

Fragment 21

```
IF YY = 92 OR 96
  MOVE 29 TO MD(2)
END-IF
```

Fragment 16

```
*       CSECT
*       XM095   ETERNAL CALENDAR 1901 - 1999
*       INPUT PARAMETERS
*             P1: DATE  TT,MM,JJ   6 DIGITS
* This calendar assumes that
*       1. january 1901 was a tuesday

        LH      R1,YYB          -- YEAR
        BCTR    R1,0            -- MINUS 1
        MH      R1,=H'365'      -- GET DAYS
        ST      R1,YTT          -- SAVE IT

        XR      R8,R8           -- CLEAR
        LH      R9,YYB          -- YEAR
        LA      R1,4            -- DIVISOR
        DR      R8,R1           -- YEAR / 4
        LTR     R8,R8
        BNZ     D1              -- NOT LEAP YEAR, JUMP
        CLC     MMB,=H'2'
        BH      D1              -- MONTH > FEBR. OK
        BCTR    R9,0            -- SUBTRACT 1
D1      EQU     *
        ST      R9,STT          -- SAVE CORRECTION
```

Figure 1: Part of an IBM assembler program for computing the day of the week. It contains a leap year subpart dividing just by four. *Note:* This fragment is based on code shown by H. Sneed, Proc. 3rd WCRE, Monterey, 1996, IEEE.

You may still experience difficulties in raising Year/2000 awareness in your organization. In that case, it may be an option to show some of the leap year code fragments listed above, or perhaps fragments used in the sources of your own organization. People will be able to see that these fragments really will cause problems. Once they understand that, they will see that date-related code in general may cause major trouble as the year 2000 is approaching.

Tool Support

To date, no tool exists that can automatically detect all fragments listed here and classify them into correct and incorrect leap year operations. The best possible route seems to be through the use of *program plan recognition* technology, as explored by [2]. This approach advocates capturing typical computations, such as determining a remainder, checking for zero, checking for divisibility (by 4), checking leap years, etc., in a library of so-called *plans*. A *plan recognizer* can efficiently check whether a given program contains code written according to one of these plans.

From the list of fragments discussed here, an initial set of plans can be composed. The better the plan recognition technology, the fewer plans will be needed to represent all fragments. For example, if the recognizer is aware of the laws of De Morgan, it will not need separate plans for both Fragments 2 and 3.

The list of leap year fragments helps to distill what challenges should be addressed by such tools:

Reduction of syntactic variety, for example by selecting certain "canonical" constructs only.

Reduction of dataflow variety, for example by eliminating transitive dependencies.

Reduction of control flow variety, for example by goto elimination.

Searching for patterns combining syntactic, dataflow, and control flow constraints;

Matching modulo a set of equations, for example expressing associativity and commutativity of logical AND.

Organizational Impact

The list of 21 small code fragments illustrates how a seemingly simple operation has been encoded in many different, and sometimes very creative, ways. Not all fragments are incorrect, but a significant number of them is.

Moreover, we have seen that many companies do not have a single library function for determining leap years. Instead, they use many different ones, even within single applications.

As a consequence, there are many procedures that might fail to compute leap years correctly. As we have seen in the New Zealand aluminum smelter example, the damage caused by such failures can be substantial, and potentially life-threatening.

This has direct implications for your organization. The tools you use may or may not be able to mark fragments as leap year related. Make sure you are aware of your tool's capabilities and limitations. Many tools simply search for lines of code containing divisions by 4, 100, or 400, or indicative constants such as 366, 28 or 29. This will indeed find all of the fragments listed, correct as well as incorrect. However, it will find a substantial number of fragments that are not leap year related — for example quarterly payment computations also dividing years by 4.

To date, very few tools, if any, will be able to split the set of fragments found into correct and incorrect leap year operations. Therefore, manual intervention is required to check all leap year operations. Unfortunately, determining that such fragments as nrs 13 and 10 are correct and that number 18 is not may be time consuming and error prone. Furthermore, unless action is taken to prevent this, programmers may still be unaware of the exact rules for determining leap years.

Since correction will rely on manual modifications, thorough testing of leap year operations will be required. Year 2000 test cases to be included are checks that February does have 29 days, that March 1st is a Tuesday, that the whole year has 366 days, and that the systems can handle 31 December (day 366) as well as the transition to January 1st 2001.

Luckily, there may be a ray hope. For most of the systems, we have 31 `C` 28 `D` 59 extra days to solve the leap year problem. That is, assuming we have time available in those first 8 weeks of the year 2000. We might also be too busy making emergency repairs to stop the two-digit crashes. Perhaps we should therefore begin straight away, and make sure our systems start recognizing the year 2000 as a leap year today.

About the Author

Arie van Deursen PhD is a project leader and researcher at CWI, the Dutch National Research Center for Computer Science and Mathematics, in Amsterdam. He has been a consultant for several Dutch banks and software houses, in the areas of software renovation and advanced software technology. He is particularly interested in tools for automatic Year/2000 remediation. Email: arie@cwi.nl.

References

[1] A. van Deursen, P. Klint, and A. Sellink. Validating year 2000 compliance. Technical Report SEN-R9712, CWI, 1997.

[2] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*. IEEE Computer Society, 1997.

[3] B. Ragland. *The Year 2000 Problem Solver: A Five Step Disaster Prevention Plan*. McGraw-Hill, 1997.

[4] H. M. Sneed. Encapsulating legacy software for use in client/server systems. In *Proceedings Third Working Conference on Reverse Engineering; WCRE'96*. IEEE Computer Society, 1996.

[5] J. Towler. Leap-year software bug gives "million-dollar glitch". *The Risks Digest*, 18(74), 1997. URL: `http://catless.ncl.ac.uk/Risks/18.74.html#subj5`.