Nevertheless, Feb 29 and March 1 are easier target dates than Feb 1, March 31 or Apr 1 or July 1. I suspect rare longer-period jobs (monthly, quarterly, etc.) will show one of the highest ratios of in-field downtime to pre-remediation bugs, simply because most organizations have had to focus testing effort around Jan 1 and Feb 29. Often putting expensive test systems through the many other conceivable failure dates simply could not be cost justified.

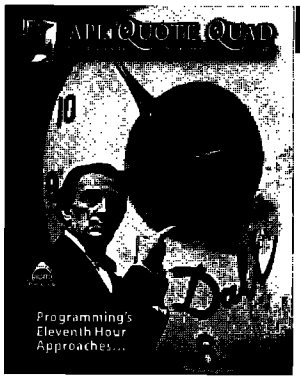
At last, perhaps sometime in 2001 or 2002, most of the Y2K bugs will have been found and fixed. At this point we can begin thinking about the 2011 bugs, the 2038 bugs, and so forth. It seems absurd to think that the versions of Microsoft Excel which incorrectly represent dates past 2011 will still be in use twelve years from now, but it seemed absurd to us two or three *decades* ago that any of the mainframe systems of the time would see the millennium.

Most absurd of all is to think that the new, modern, replacement languages and operating systems of the present have similar structural date bugs built into them. Yet quite a few flavors of C, Unix, etc., have dates represented as the number of seconds since an arbitrary date in the 1970's, stored in a four-byte integer. Most of these overflow some time around 2038. I hope to live long enough to laugh at the hapless multitudes trying to find and fix all the toasters with faulty C code in them before 2038.

Last, and perhaps least believable, will be the failure in the second half of the 21st Century due to our Y2K bug fixes. Many Y2K bug fixers, myself included, have been guilty of using a fixed windowing or pivot date to solve century bugs. That is, we kept the two digit year, but added code that said "*yyyy*← 1900 + *yy* + 100 × *yy*<50". This treats "50" as "1950" but "49" as "2049". Various pivot dates have been used, and one can only hope devoutly that half a century is long enough that all of this "fixed" legacy code will be on the junk heap before the window expires. ■

***Clement Kent** has been a happy user of APL since being exposed to it in high school in 1969. He consults in the areas of software engineering and project management in the financial services sector, as well as the application of software to computer integrated manufacturing for semiconductor companies. For the last three years his primary involvement with APL has been in leading Y2K projects. He is currently on the Executive Committee of the Toronto APL Special Interest Group, and organized the November 1998 symposium "Challenges in Medical and Biotechnology Software" which highlighted some commercial APL products in this field. He may be reached at "clementk@acm.org".*

Illustration by Jan Hicklew



COVER STORY

Y2K and APL—An Overview

*—by **Clement Kent***
Godel Computer Solutions Ltd.
clementk@acm.org

I HAVE WORKED ON Y2K PROJECTS IN APL since July of 1997, which makes me a sort-of-expert. My goal in this article is to help those of you still in the throes of Y2K work, and to give the rest of you—those who have completed your Y2K work—a basis for evaluating your own projects. Of course, if you haven't yet started your Y2K project, you're doomed, and you may as well stop reading and take up dental floss farming in Montana.

My Y2K experience has been with large applications on mainframe APLs, although I have experimented with some vendors' PC APLs to understand the issues there. Whether you have a large or small application, on mainframe, mini, or PC, you still need to understand some Y2K issues common to all these environments.

Is Y2K a problem for APL?

In a word, *yes*. Why?

The real answer is, *human nature*. Although I have worked on Y2K for a subjectively very, very long time now, and am quite sensitized to its issues, when I make notes in my personal diary I still write "meeting Jan 27 99" and the like. Writing the '19' is just too tedious.

You will read many articles that explain in excruciating detail how expensive memory or disk was in the 60's or 70's or 80's (note we don't say "nineteen eighties," do we?). But programmers, even quite good ones, continue to create code which leaves out the century, even in the 90's.

In one customer's Y2K project we looked at a number of large applications, none more than ten years old. Memory or disk wasn't a real issue for these programmers any more. Yet, almost all of these applications had some problems when placed on a test machine which was advanced to 01/01/2000, and several of the most critical ones flat-out died.

A second, darker side of human nature provides an equally important reason for doing your APL Y2K project. This is the human need for a scapegoat when things go wrong. In some countries, especially the United States, lawyers and the media facilitate this need and earn big bucks in the process. What they earn, you or your customer pays for. So even if you have strong reason to believe your application never ever, honest to Bob, used any non-compliant code, you'd better prove it prior to the big '00'.

Legal issues

The previous paragraph segues into this section: assume that if anyone other than you uses your software, they'll try to sue you for any bug (whether caused by Y2K or not) that happens between about September 1999 and April 2000. Even if their suit is groundless, you need to be able to defend yourself at minimum cost. To do so, you need to understand how legal reasoning and rules of evidence differ from what programmers use.

In addition, non-legal commercial issues demand that you do the same prep work needed to face down lawyers, so that competitors, customers, regulators, and media (CCRM) folk have no reason to condemn your software in advance. You can earn this condemnation simply by doing nothing. The pervasiveness of Y2K problems is such that doing nothing is taken, rightly or wrongly, as a statement that either your software is toast, or you're an idiot, or both.

As a result of the need to reassure the CCRM folk that you're on the case, you'll find that most companies with a high public profile are posting web pages with statements about Y2K compliance. These statements are very necessary to prevent a stampede to your competitor's software, *but* they represent an assertion that a lawyer can use against you if problems do arise in 2000 contrary to what you implied... so be very sure that what you state about compliance is true.

An interesting example of this from the non-APL world is Microsoft. In June 1998 the Microsoft marketing department billed Windows 98 (note: *not* Win 1998!) as the solution for all the Y2K ills of your PC operating system. This pretty well guaranteed that corporate MIS departments under the gun to show Y2K compliance internally had to update all their 95 and 3.1 machines to 98, so it was a great marketing manœuvre.

This article is typed on a new computer, with Y2K-compliant BIOS, running Win98. The very first patch I had to download from Microsoft was a Y2K patch for Win98!

The moral is obvious. You've got to prove your software is Y2K compliant to the world soon, and you've got to be confident you're right. Mistakes will be embarrassing and costly.

You have to prove it works, a.k.a., "Testing"

Obviously, then, you've got to have something beyond a bald-faced assertion your code is compliant. The best thing to do is find out what's wrong with your code, fix it, and then test the fixed code.

Of course, you may not think anything is wrong with your code. Congratulations! You've saved yourself 25% of the work of a Y2K project! All you have to do now is the remaining 75%, the testing.

You have to convince a lawyer with your proof

So, you changed the system clock, loaded the ws, and it ran just fine. You're done, right?

If you work for yourself and no one else uses the code, yep. Otherwise, so sorry, nope!

Lawyers have a funny way of reducing your credibility on the witness stand. Unless you have documents showing what you tested, how, when, and what the results were, you will be exactly the kind of "sacrificial victim" every lawyer dreams of if you are unfortunate enough to end up in court.

Thus, when you test, you will want to do printouts, screen captures, and probably annotate the results in some way. You need to make sure all such documents (whether paper or electronic) are dated in some way, and if possible signed by the person who did the tests. Finally, you'll need to ensure that the documents are stored safely for a number of years (at this point, at least three to four years) in such a way that you could retrieve them and make sense of them if you needed to.

Ideally, if you have the time and money to do it, you should have not just the results of the tests documented, but the *test plans* by which the tests were guided should be written down, reviewed, signed off, and archived.

Last but not least, you will need some way of distinguishing the code which has been tested and is known to be correct from earlier versions. You need to record *what* you tested in some detail. Imagine you are in the witness stand. A lawyer for the prosecution says, "so, Ms. Jones, you say you tested version 1.2.3 of the software. How then do you explain its wiping my client's hard drive?" You must be prepared to explain how to show that the client was actually using version 1.0.0, not 1.2.3.

If you work for a large, publicly exposed company it may well be necessary to consider embedding a version number as a comment in each APL function, or using a similar scheme. It would be very easy for a co-worker or customer to accidentally mix versions together; how are you going to show that this did or didn't happen? The phrases "code inventory," "code bill of materials," "version control," etc., crop up often in the Y2K literature.

Advantages and disadvantages of an interpreted language

Given that perhaps 75% of project cost for Y2K is related to testing and code inventory activities, how does APL's ease of use and debugging help or hinder this process?

"Test, then fix"

I have seen small APL Y2K projects run as follows:

- put the code on a test machine
- run it in the year 2000
- look for things that break and fix them when they happen

This approach is totally antithetical to how most large Y2K projects are run. Most pointy-haired managers familiar with C, Cobol, and other compiled languages would be horrified if this were proposed to them. The cost of debugging, fixing, and rebuilding is so high in such languages that the strategy is impractical.

Is it practical in APL? Sometimes!

For smaller applications without too many interactions with other systems, this "test, then fix" strategy can work very well. It reveals many problems quite early on, rather than waiting for them to be discovered at the last moment.

The risks inherent in this strategy are real, but can be managed with some discipline and care:

- risk that fixes will not be made everywhere
- risk that everything won't be tested
- risk that traceability of testing and fixing will be lost

Due to the ease of real-time debug and fix in APL, many programmers are used to a rather casual approach to fixing: change the line, →□*LC*, keep going. This is fast and convenient, but you must be very careful to ensure that all of the fixes are carried back to the main working version of the application, wherever that is.

Informality and ease of use is why many of us fell in love with APL, but that is not necessarily compatible with achieving full test coverage of all function points in an application. Testing is best done from a test plan. This may be as simple as a numbered list of things to exercise, or it may be as formal as an entire test suite with drivers, test data, scripting tools, and output logs. The temptation in a small informal project is to dispense with all of this and "just do it!"

If the testers are disciplined, understand the application and APL very well, and don't have too complex a system to deal with, the inherent swiftness of the test, debug, fix cycle in APL can compress the time to getting a fully working Y2K compliant application.

If the code is entirely for in-house use, is not business critical, and is well understood, this strategy may be justified.

Needle in a haystack: Fixing non-compliant APL code in large systems

If "test, then fix" works so well for small APL systems, why does it fail for large ones? For, fail it does.

Here again the answer is, surprisingly, that APL is interpreted. This means that each working copy of a function is also a clone of the code. Many of us who've worked on large systems have been frustrated by the re-occurrence of a bug fixed weeks or months ago in some other copy of the function we didn't know about and didn't fix.

As the number of programmers and testers involved in an APL Y2K project rises significantly past two or three, the risk of not communicating the need for fixes rises geometrically. The work must be segmented, but the fixes cannot always be isolated.

One of the mantras of MIS managers for the last decade has been "code reuse." Most large APL systems have at least some facilities for encouraging reuse, such as central libraries of utilities, centralized function paging files, or similar code control systems. Nonetheless, the fact remains that there is always the possibility of caching old copies of function in workspaces or files outside of code control systems or public utility libraries.

When one of these multiply-cached utility functions has a non-compliance problem, a potentially large task arises: find all the copies of it and change them all. This task crosses subproject boundaries and needs to be communicated to many programmers. It is common to find that as the utility was copied, it evolved from its original form, so that <daynum> in ws "123 datefns" is now subtly different from <daynum> in "3456789 bondcalcs".

This problem of dealing with family trees of cloned function copies is best approached through use of tools (discussed in a later section). It is above all a management problem that deserves careful thought from the beginning of a large Y2K project, and constant reexamination.

A simple example makes this clearer. Suppose we have a function <daynum> which has as argument (year, month,day) triplets, and returns an integer Julian date. We'll assume that the version of <daynum> currently in the utility libraries handles Feb 29, 2000 correctly, but that an older version didn't. Furthermore, the older version assumed that any year less than 100 should be "windowed" to be in the 20th Century, while the new version applies the rule that if the year is two digits and less than 90, it's in the 21st Century (a 1990–2089 window).

Your job, as Y2K project leader, is to search out all of the old <daynum> clones, including the ones that were renamed <datenum> or <foobar> for perverse reasons, and replace them with the new version. This has been well explained to your programmers and they are fairly careful about doing this.

One day, one of them comes in and says "our portfolio system has two-digit years going back to 1955. I want to replace the <daynum> 1990–2089 window with a 1950–2049 window in this system." What do you do?

- If you say yes, you'll get the portfolio system compliant quicker and with less fuss, at the cost of perpetuating functionally different clones of <daynum>.

- If you say no, you'll either have to convert all the dates in the portfolio system to four-digit years, or condone the use of some utility other than <daynum> for calculating Julian dates.

In the best of all possible worlds, you will have time and budget to say "dammit, let's convert all the years to four digits." If you lack time or money, you'll have a tough call to make.

The worst case of all is the most frequent one in the APL world. The programmer isn't used to coming to you for trivial little problems like that, so they automatically chose a solution they liked. If this involved keeping <daynum> but changing the window, what happens if the "bondcalcs" system and the portfolio system call each other? Will 30-year bonds maturing in 2005 have an issue date in one system of 1975 and in the other, a date of 2075? What will happen to Disney Corp 100-year bonds?

This potential for chaos caused by loose code version control in APL, and by the otherwise admirable tendency of APL programmers to "just do it" is a bigger risk for large systems with shared code or data, than for small, independent systems.

How to manage an APL Y2K project

Many tasks in an APL Y2K project are the same as for other languages. This doesn't mean they're not important; only that you now have a plethora of places to read about them. This is the longest part of this article, but probably the most important. I can see the programmers' eyes glazing over already.... Management tasks include:

- Have a strong management structure
- Make sure you have full support from the CEO
- Engage in Triage
- Build test facilities and test plans
- Get and use fault-detection tools
- Estimate effort from the code base
- Plan for Code Release and Control

Let's look at these in turn.

Have a strong management structure

Y2K projects have a large scope (all the code you use) and a fixed deadline (August 1999 or earlier—I'll explain that date later). Large software projects notoriously fail to meet deadlines. Why should Y2K be different?

The best solution to this problem is, quite simply, excellent project management. Without this, unless your project is very small, you will have nasty surprises. See you in court!

In many large organizations, departments and different divisions don't always work well together. Unfortunately a Y2K software project often crosses several departmental boundaries. Strong management is needed when department heads find reasons for denying you critical resources or facilities. If you can't get them to cooperate, see you in court!

Strong management is needed to limit scope. Because you must look at all the software, in most big companies, you'll see an awful lot of crap code. The temptation is great to "fix" it while you've got it opened up for Y2K. Managers must say "No!" often. If you don't, you can explain why you didn't finish to the lawyers in court!

Many companies no longer have a strong middle management layer due to downsizing and reorganizations. Many managers in software companies were promoted from the ranks and are stronger at coding than at Gantt charts and task estimates. Your company may well need to consider hiring extra project management staff for Y2K. The cost of this can be as great as or greater than the cost of coders and testers, because the PM's are in greater demand. Can't spend the money? See you in court!

Make sure you have full support from the CEO

So, you discovered that you need to spend megabucks that weren't budgeted for PM's and test facilities, the project is already behind, and the head of Customer Support has flatly refused to lend you a dozen customer support reps for product testing. What can you do?

A) Quitting is one answer. It is one frequently employed by those higher up the ladder in the Y2K project, especially by those intelligent enough to see the chopping block being prepared for them in the Boardroom. This is not a joke! On the hairiest Y2K project I worked on, the Program Manager (boss of the Project Managers, the Big Cheese) left to take on a lovely new job before the first application got to the test-bed. Turns out he'd been looking for a new job since learning that he'd be "Lord of the Y's," as PM's sometimes refer to each other on these projects. The interregnum while we waited for a new Big Cheese set us back considerably.

B) Perhaps the only other good answer to the "what can you do?" question is "go to the top." The CEO of the company *must be totally supportive* of Y2K project needs. If they are not, see (A), above. The hairy project was saved, in large part due to a high level of upper management commitment, both in dollar terms and in willingness to bang heads, re-prioritize other projects, and in general, *lead.*

Engage in triage

If you are reading this in Quote Quad in 1999, and your Y2K project hasn't nearly finished, *you* are nearly finished! Even the smallest projects have unavoidable time lags in them; when these are additive, as some are, schedules slip rapidly. It is very unlikely that any large Y2K project starting in 1999 will finish in time.

A good finish date to have set for yourself would have been Dec. 31, 1998. As you're already past this date, the latest you can consider is August 1999. This date is chosen for several reasons:

- Some COBOL and related legacy apps will begin failing in September 1999. Your code needs to be done, and visibly done, to the world at large before then. If not, the CCRM gang will assume you're in *big* trouble. They'll be right.

- You will have unanticipated slippages. You need to allow several calendar months of contingency to deal with these.

- You will have to spend time in Q3 and Q4 1999 installing other companies' newly Y2K compliant software versions and retesting your own code after their changes are in place.

In many companies there simply won't be enough people, dollars, or time to fix and test all software before August 1999. The answer in this case is to engage in *triage:*

- ***The A List:*** Applications that are business critical. Do everything humanly possible to fix or replace these.

- ***The B List:*** Applications that are important, but not time- or business-critical. Do what you can to fix these, but make contingency plans to fix them on or after 01/01/2000 when and as they break. A List jobs pre-empt those on the B list.

- ***The C List:*** You can't afford to fix these. Make plans to get rid of them before the new century arrives. Note that this often takes work that must be budgeted for. Be prepared to find in 5–10% of the cases that the cost of removing a C-List app is higher than the cost of fixing it, in which case your B List just got bigger.

You, the Y2K project manager, have a big task doing triage. You must persuade upper management to kiss the systems on the C list goodbye. You must dissuade departments from embarking on upgrade or enhancement projects for applications on the B list. You must ensure the A list is small enough to really get done by August 1999. Good luck! See you in the CEO's office....

Build test facilities, test plans, and test data

Whole books could, and have, been written on this. Here's an abstract of the problems:

Test facilities

Setting the system clock forward is easy enough for PC applications, and can be excruciatingly difficult in some mainframe and mini O/S's. If you work on big machines, be prepared to spend an unconscionable amount of time fighting the O/S, disk and tape management utilities, and third-party software when your test machine is moved forward and backwards in time. This adds big costs, and big inherent time lags, to testing schedules.

One customer's IBM mainframe APL system was operated for them by a service bureau. Before a test machine could be set up, the service bureau required several months of preparation to isolate the test machines from the other production environments. IBM had warned of catastrophic results if the O/S saw files with future modification dates, so test data sets had to be carefully segregated behind operational and technological firewalls. Although the customer's Operations Department did an excellent job of planning and preparing for this, it took over six months to get the testbed environment fully operational.

Once operational, scheduling flexibility for testing was limited by the need to fully restore the system from tapes every time the clock needed to be moved back at the end of a test cycle. This meant that the best that could be achieved was about six to eight hours downtime between cycles. This prevented testing on more than one critical date transition on any given calendar day, in most cases.

We felt a little bad about this "batch mode" environment until we heard about another division of the customer's company which had much more modern, cutting edge hardware and software (not APL). That division was rarely able to reset their test machine to the beginning of a cycle in less than several days effort, proving that our experience was not unique.

Those of you with networks of PC's are probably grinning in a self-satisfied way at this point. While individual PC's can be reset fairly easily, networks of them are a different matter. This is due largely to application software checks on different boxes that look for synchronization across the net. This is most apparent in near-real time applications such as trading systems. Thought must be given in these cases to creating an easily managed test network which can be quickly resynchronized to any specified testing date and time.

Test plans

The larger the system and the more critical it is, the more important it is to have a written, well organized test plan. It is very difficult to write a good test plan unless you understand the application reasonably well. Therefore, test plan writing will often bottleneck on scarce resources: the application experts.

If your company already routinely writes and executes test plans, estimating this task for Y2K will be relatively easy. However, many departments with APL systems have been able to muddle through in the past without disciplined testing procedures (the 'if it breaks, we'll fix it' school). In one organization we had to bootstrap testing skills from a small core of early introducers to a much larger group of application experts. This was time consuming at first and often surprisingly painful for the experts, most of whom derived their job satisfaction from writing code, *not* from test plans. It was not unusual to find that the first plan written by an individual took four weeks or more, while subsequent plans took progressively less time.

In addition to the time it took an expert to write a test plan draft, we had to plan for and allocate time from several other staffers to participate in test plan reviews. Many experts felt this was an unmitigated bit of bureaucracy, until they found fundamental flaws or lacks of coverage being pointed out by their peers in the review meetings.

A second, very valuable role of review meetings, was to limit the scope of plans. Some experts, once they get the bit in their teeth, will write overly extensive plans. Project management and their peers must pare these back to just the minimum tests needed.

In one Y2K project, the PM decided to defer reviews to a later point in the project. This project ran into serious difficulties during the actual test cycles. Review and revision of test plans as early in the project as possible is extremely important.

Test data

It is easy to overlook just how costly and difficult it can be to generate good test data. The usual assumption made is "well, we'll just take a copy of the production system's data and move all the dates forward to Jan 1 2000, then test." This rarely is the whole story.

First, it's quite likely that your test machines will be smaller and less powerful than your production machines. You'll have to prune your database rationally to fit it onto the testbeds.

Second, most large systems consist of many interacting subsystems. These rely on chunks of data from each other for their normal functioning. For example, there might be a master table of insurance policy types and identifiers. Many subsidiary systems will refer to the master table. If data needed in the policy renewal subsystem is not available because it was pruned out of the master table, spurious errors will be seen.

In addition to data integrity, date consistency is often an issue. Business applications often "know" which days are business days and which are not; they behave differently on holidays and weekends. Simply grabbing data for the month of December, 1998 and changing the year to 1999 won't work because the weekdays are different. In addition, January 1, 2000 is a Saturday. Testing business applications only on January 1 won't work because they will only exercise weekend and holiday behaviour. Tests will need to include data for January 3, and in some jurisdictions this too will be a holiday, so January 4 data becomes a requirement. Similar problems arise around the leap date.

New test data will need to be created in some cases. Understanding the business rules used by the application to validate such data is important. For example, in one bond pricing system, prices were not updated unless they had changed, so test data had to incorporate price changes. However, some bonds were priced in dollars and cents, while others were priced in dollars and eighths—the test data had to follow this pattern. Last but not least, applications often apply "sanity checks" to input data. These will only be triggered if prices move too much in either direction. An initial test data set was completely rejected, leading to a massive exception report file and a disk full, because the programmer had not understood the sanity checks within the application and had simply added $10 to all prices.

We often focus first on production data: the actual working data of the applications under test. But configuration data also needs to be ported to the testbeds, and frequently will need to be revised there. Tables of account names, access privileges, mail account ID's, etc., will not transfer without change.

For instance, one customer's security policies required that account passwords on testbeds and development systems be different from those on the production systems. A legacy system which had been built in place on the production system had never been ported to another machine. We found out the hard way that passwords were in some cases hard-coded into locked functions. In some cases no unlocked reference copy of such functions could be found. This is much like the case of the COBOL system where only object code and executables exist, the source having been lost! In this instance testing had to wait for the availability of a few trusted senior staff who had permission to unlock functions, remove sensitive data, and pass the unlocked, desensitized versions on to testbed building staff. Delays resulted.

In another instance an error-condition reporting utility was shared by almost all major applications. It had privileged access to the company e-mail system so that problem reports could be sent in real-time to support staff. Naturally, such a sensitive system was locked up and closely guarded. This had been anticipated and unlocked, desensitized versions were already available when we went to build the testbed. What we had not anticipated was that destination e-mail addresses would be hard-coded into some calling routines; again delays resulted as this was remedied. The risk in such situations is delivering spurious error reports from the testbeds to the production support staff.

In yet another instance, a utility system used for managing user access to production systems had been built many years before. It had been created in part by staff knowledgeable about the internals of the APL interpreter, and made significant use of privileged I-beam functions. Most of the code was locked to hide such mysteries from mortal APL programmers, with the result that a junior, well-meaning staff member misunderstood the system setup and accidentally trashed large parts of the entire testbed user access files—a mistake that cost over a week to remedy.

These funny little pratfalls do not make the PM any happier when they happen.

Get and use fault-detection tools

The Y2K literature is full of advertisements from tool vendors. Many of these tools will slice, dice, and cook your code for you. I do not believe the APL marketplace is big enough to justify the large upfront investment to build tools that can actually change code or data in APL. APL is too rich in idioms and dialects for this to be very effective.

However, there are tools on the market which are essentially database search engines, in which all of the code on your APL system is the database, and the patterns searched for are code fragments or utility functions known to be associated with date logic. For help in locating and evaluating such tools, talk to your APL vendor.

Such "fault-detection tools" have three essential components:

- a set of code patterns to search for
- a search engine
- a reporting system to record matches to the patterns

Ask the tool vendor whether the set of search patterns can be easily customized at your site. Use of the tool is best thought of as a recursive process: use it once, along with your experts' knowledge of your systems, to locate any commonly used utilities and code patterns that correlate well with date logic or data. Then, add the names of these utilities, or a description of the code patterns, to the set of patterns searched for and run the tool again.

Some patterns are largely independent of APL dialect and should always be searched for:

```
'19'
19+
1900+
⎕TS
100⊥
100⊤
100|
```

Note that the last four patterns will be found many, many times in quite innocent code. It seems better to allow the many false positives caused by such patterns than to risk missing date code by excluding any of them.

It is arguable that any identifiers containing the strings 'year', 'date', 'dt', 'ymd', 'month', 'mth', 'day', and 'dno' should be reported by the tool. Again, many false positives will result.

The search engine should incorporate knowledge of the syntax of your APL dialect. It is important to be able to specify whether a given pattern is matched only when it is a valid APL syntactic token, or whether it can be matched anywhere—for instance inside of quoted strings, comments, or as a substring of other identifiers.

It seems obvious, but you should ask: can the search engine *find* all the code on your system? Is it fast enough? Smart enough? APL functions may be present as functions, as vectors, as character matrices, or inside of dialect-specific constructs such as packages, namespaces, nested arrays, etc. On a large APL system, can the tool search the whole system in a convenient length of time? One tool I encountered took from late Friday night to late Sunday afternoon to scan a full production system. Had it been 25–30% slower it might not have been usable.

Thinking ahead to the end of your project, will the tool allow you to somehow designate some functions as "Okay—don't search me"? As your project progresses, you'd like to mark functions known to be compliant, perhaps with a special comment, or perhaps by entering them in a database. Of course, other functions with the same name but different contents must still be searched. Can the tool make this distinction?

The reports generated by the tool are essential to your project. At a minimum they should unambiguously identify the functions with pattern matches by name and location. It is useful if information about *which* pattern(s) were matched is reported. Some indication of function size, such as number of lines or number of bytes, can be useful for estimating work to be done.

This reported information will ideally be available to you in machine-readable format. APL character sets are notoriously unreadable from within Excel or Oracle, so find out whether the data will be APL-legible.

Estimate effort from the code base

One large challenge for the project management staff and application experts is estimating the amount of effort required for Y2K work. The problem is that most of the staff have never done this, aren't familiar with what it entails, and so have no basis for comparison to past projects.

For large systems, some effort figures can be derived from counts of the code base. This rather mechanical method isn't great, but it's a lot better than nothing.

The first step is to run a fault-detection tool and look at the number of functions it finds with pattern matches, and their numbers of lines of code. An initial crude estimate can be made for the "code assess and fix" phase, during which suspect functions are analysed and if required, changed, by assuming that average APL contractors will be able to handle about fifty functions per month during the first four to eight weeks of work, and about 100–150 functions per month thereafter. Full-time staff working in an application with which they are already familiar will move to the higher figures more quickly, but non-project lost time—reading e-mail, attending meetings, vacation and sick time and training courses and time cross-assigned to other projects—often reduces the effectiveness of the full time employee below that of the contractor, simply because they cannot focus exclusively on the Y2K task the way the contractor should (hint: don't hire contractors for less than full time, lest they end up like employees).

Writing test plans and building test data is harder to estimate. I've already mentioned that when an employee first starts writing test plans, allow a month per plan for learning curve time. After that, the time to complete a plan is largely dependent on application size, complexity, and use of date logic. These three factors should be estimated for each application, and an estimate derived from a set of weights (a.k.a 'fudge factors'). Closely monitor actual effort versus estimated during the early phases of the project and you can adjust the weights as you gain experience.

A sanity check is that test plan and data generation should not take less time than the code assess and fix, nor more than twice as much time.

Actual testing, if the test plans, data builds, and code fixes have been carefully done, can be rather slow, but low effort. Expect two to four passes through each test plan before it all works. Often elapsed time for these two to four cycles is a bigger schedule factor than the effort expended. For instance, if a bug is found on a Jan. 4 2000 testing date, but there is only one Jan 4 2000 date available every two calendar weeks on the testbed, successful completion of the test will take four to eight weeks of elapsed time.

An important tactic is never to start formal testing on an application in the testbed until an application expert has 'hand tested' the application installation. In my experience many more configuration bugs are found on the testbed than Y2K bugs; these should be cleared out before formal testing starts, or you will lose test cycles.

Plan for code release and control

When beginning a Y2K project we tend to focus on the immediate job at hand: finding the code that's wrong and fixing it. We tend to forget one potentially large and ugly problem that arrives near the end of the project, which is releasing the fixed code.

Because the scope of the Y2K project is so large, there can be many interdependencies which have to be resolved or taken into account in the release schedule for fixed applications. Operations personnel tend to be rightfully suspicious of statements such as "we plan to release new versions of fifty major applications next Sunday." What will you do if they insist that no more than one major application be released per weekend/holiday? Do you have time to complete all releases before the year 2000?

If application A has changed date formats internally, will apps B, D, and F which use it have to be released simultaneously? If utility <daynum> has been changed to window two-digit years, and is paged from a central function file, do you know all the applications which use it and whether they need to be upgraded at the same time or not?

If your application produces data used by third parties or customers, and date formats have been changed, have you provided them with test data sets in the new format? Have you negotiated a mutual upgrade date, or will you have to operate the old and new versions in parallel for a time while they upgrade their systems?

Finally, have you taken into account any parallel development going on simultaneous with Y2K work? Are all bug fixes and other upgrades up to date in the Y2K compliant version so there will be no regression of functionality?

An example of the kind of bind PM's can get into was felt at a customer who did a significant amount of business in Europe. As a result, all financial systems had to be upgraded to deal with the Euro currency prior to Jan. 1, 1999. This was the top company priority, and the Euro conversion project outranked Y2K. The Y2K last release date had been set during 1997 as Dec. 1, 1998, assuming a one month freeze prior to Euro conversion was desirable. In the event, the Euro staff cautiously negotiated a longer freeze period prior to the Euro Big Bang. An unexpectedly large number of Y2K release dates had to be moved to the beginning of November, putting a real squeeze on project staff at a difficult time.

Who can find and fix and test Y2K bugs in APL code?

While most APL applications can't be shipped out to Y2K conversion factories, due to the small market size for APL and the many dialects, the APL Y2K project is not doomed to use only full time, salaried staff. In the early phases of the project, quite a bit of flexibility can be gained by using contractors for code assessment and fixing.

Although contractors will cost real, tangible dollars, during the first half of the project they can really accelerate progress. This is because, in conjunction with a good fault-detection tool, a very large percentage of Y2K bugs can be found by any reasonably-competent APL programmer. Actual experience from the past two years suggests that, with good management, contractors are able to find and fix 99% of Y2K bugs in applications with which they are not initially familiar.

The primary requirements for assessing and fixing Y2K bugs are (a) high level of APL competence, and (b) high tolerance for drudge work. Unfortunately, (a) and (b) don't always go together. This puts a premium on careful interviewing, reference checking, and monitoring of contractors throughout the project.

In one large Y2K project, I diverted a few contractors to build a "compliance database." This took the output of the fault-detection tools as its raw data, and was then updated by each contractor for each function that they had examined. This proved extremely useful. I was able to monitor each contractor's performance over time (the 'Big Brother' function) and report to the customer's VP in charge on a frequent basis on progress per dollar billed. Once contractor staff were through their learning curves, I was able to predict quite accurately when the assess and fix effort would start winding down.

Thus, I would have no hesitation in suggesting the large-scale use of competent APL contractors on Y2K projects, so long as good management oversight and guidance is provided to the contracting staff.

Testing is an entirely different kettle of fish. It requires a much more detailed knowledge of how an application works. A randomly chosen, APL-competent contractor will require two to four months to become familiar with a large business application to the point where a test plan can be written. If sufficient full time staff with expertise are available to write test plans, this will almost always be more cost effective than using contractors.

Unfortunately, in many organizations the extra load of a Y2K project cannot be resourced from full-time staff, who may be allocated to business-critical support and development. In this case, strategy is important.

In the large project referred to above where contractors were used in the assess and fix phase, each contractor was assigned to fix several different applications (sequentially). Several contractors were working in parallel at the same time on certain large applications. This "wide but not deep" approach was used

because it was felt essential to get certain apps fixed early, as they were used by many other apps later in the project.

This goal was achieved, but as a consequence of being reassigned every month or two to a new application, contractors were not able to gain an in-depth understanding of any one system. Had we instead assigned fewer programmers to each application, but kept each one working on that application afterwards to assist in test plan and test data building, we would have built expertise much faster, at the expense of finishing the fix phase for some early systems more slowly.

Our plan was to have only full-time experts prepare test plans and data. However, due to turnover, sickness, and higher priority projects, we did not receive the allocated effort from full timers. This forced us rather late in the game plan to switch back to using contractors, without having built their expertise in advance.

In summary, then, if you are quite sure that enough full time staff will be available to write test plans and build test data, restrict use of contractors to the code fix process, and let them go wide, not deep. But, if you have any doubts about the number or availability of full time experts, choose some contractors (your best ones) , and re-focus them to go deeply into a few applications, from beginning (fix) to end (test).

What to do when management suggests converting to C++

A large 1999 APL Y2K project has a high a priori probability of becoming a death march project. Under such conditions, pointy-haired managers do their best to annoy and irritate staff with stupid suggestions. This is known as "out of the box thinking" in the trade.

The most likely jack-out-of-the-box suggestion from your local manager will be that you avoid the terrible costs and boredom of doing Y2K work on your old APL systems by converting them all to C++ (Java, Oracle, OLAP, Perl, insert your buzzword here...).

Remember, physical violence, while momentarily satisfying, is not an appropriate response to such well meaning suggestions: "must...control...fist...of...death" should be your mantra at such times.

The appropriate response, which has the added virtue of being the correct one as well, is that large application replacement projects are too risky in 1999, and that they should save the budget for such a costly exercise for the year 2000. Point out that the lower debug costs of APL make it uniquely suited to lowly maintenance projects like Y2K, unlike such godlike and hard to debug languages as C++, where everything has to be done right from the start or you are doomed.

Your manager is likely to walk away reasonably satisfied with this answer. Don't worry, he or she will have been downsized or have quit by the year 2000 anyways.

My favourite Y2K bugs in APL

This section is for late at night, when the deadlines are looming....

My very favourite Y2K bugs are actually a class of bugs. They have to do with stashing dates in file names.

Now everyone knows that it's a bad idea to use two-digit years in dates. But, various O/S's put inane restrictions on file name lengths (eight in DOS/Win 3.1, eleven in older mainframe APLs, etc.). What is the poor programmer to do but to chop the century off, so that today's data can be stored in '1234567 JN980611' or 'JN980611.FOO'. Of course, there are ways of compressing the full year, month, and day into six characters, but who wants to enter a "DIR" command and see a bunch of file names like 'JNBTZ9LK.FOO'?

In my very favourite APL Y2K bugs, the system works just fine with such files during the year 2000 when creating them (but look out for that `nm←'JRNL',⍕100⊥100|3↑⎕TS`) and even when reading data from them, etc.

The thing that's so delicious about these bugs is that they wait to hit you until an unpredictable time after the new century hits. Then, after some interval, the daily (hourly, weekly, monthly, quarterly) housekeeping task wakes up. "Hmm," it says, "which files are older than my five-day (week, month, etc.) retention period? I'll delete them." As this is a very obscure and unimportant task everyone forgot to check its code, so it's not windowing its dates. "Oh look!" it says. "There are a bunch of files from 1900! I'll delete them first."

Now the funny thing is that I'm *not* making this up. In several separate projects, in code written by completely different programmers, assessed and reviewed by competent, serious contractors and expert full time staff, these bugs sailed through all the inspections to the testbed stage. There, they brought very important applications to their knees, by deleting the current day's input data instead of or in addition to last week's, or by erasing all history logs, etc.

I'm not sure what it is about these obscure, poor cousin housekeeping tasks, but psychologically they are invisible to most programmers, sort of the way the serving staff is invisible to an aristocrat. They will, however, repay your closest attention.

After these 'kill the application dead' bugs, my last example seems very modest. However, I will note it as an example of why I don't think APL will be susceptible to automated Y2K code correction any time soon. This line was observed by an alert contractor, not flagged by any fault-detection system:

```
y←'1','9',y
```

Why, why would anyone do this? Why, because `y` is a matrix, of course.... ■

***Clement Kent** is President of Godel Computer Solutions Ltd, a software firm. His work is split between the commercial APL community, where he leads teams working on the Year 2000 problem, and the CIM (Computer Integrated Manufacturing) world where he is a consulting architect for computer systems that help run computer chip factories. He has worked with APL since before he had a drivers license and hopes to do so well into the 21st century. He may be reached at "clementk@acm.org".*