Using Microsoft Access to Store Function Definitions

Dick Bowman

Dogon Research

Email: bowman@apl.demon.co.uk

Authors Note: This is another version of the document distributed at APL95; the technical material is unchanged, but the purpose is to experiment with Adobe Acrobat as a method of publishing APL and J material on the Web. No endorsement of Adobe products is intended or should be inferred.

Introduction

This paper summarises some preliminary experimentation with using Microsoft Access as a vehicle for storing APL functions definitions separately from the workspace.

The work appears to offer some advantages in terms of improved code portability between interpreters.

Some Problems with Workspaces

The only way of storing defined functions (and operators, where relevant) which is made explicit by vendors is the saved workspace. It is rather surprising that this situation has lasted so long, and some of the problems (along with some proposed remedies) were explored in an earlier paper [Bow94].

Two of the main difficulties which this project seeks to address are:

- Imprisonment. A defined function can, in general, only be saved as part of the current workspace.
- Versioning. Current APL systems do not provide automatic audit trails of changes to code.

What the vendors do provide is the means for the individual user to design and implement their own approaches to code storage.

J is an interesting exception to the rules (as usual?); workspaces have been abandoned as a storage vehicle.

Function Files

Most large users have solved the problem by implementing function file systems based around component files (for those dialects which offer this facility). Many of these come with elaborate schemes for building the active application workspace dynamically, ensuring that appropriate versions of each function are retrieved and also retaining historic records of earlier code versions.

These systems usually work very well, but remain proprietary and have not been integrated into the APL language products available for purchase. Nor indeed do they seem to be offered as a post-sale accessory.

An Alternative

At APL94 John Baker described a mechanism for storing J words externally to the J system [Bak94]; among the advantages:

Unique definitions

Complete definitions

Stored relationships

The mechanism used in this case was FoxPro as the database system with DDE used to communicate with J.

What was most interesting about the project was that J words could be accessed as things (devoid of meaning) - I would have liked to say objects, but the word is overused - and that definitions could be guaranteed to be unique.

Would a similar approach have any benefits for APL?

The tools at my disposal were various APL interpreters and Microsoft Access; since Dyalog APL is distributed with Insight Systems SQAPL/EL product and J includes an ODBC driver for Access there seemed to be enough. A parallel development was carried out with both Dyalog APL (Version 7.1) and J (Version 2.03); code for both is included.

The APL character set always provides lots of opportunity for discussion, and the first unknown was whether APL characters would survive a voyage through Access. If they did not, then the project was sunk.

The project is buoyant.

Detailed Description

What I can describe now is the very earliest usable version of the mechanism; as of the time of writing it all appears to work.

The first step is to set up an Access database and set up the ODBC driver for this source; the data source is known as CodeBank for the rest of the paper.

CodeNewTable

Sets up a new function storage table; this is a design decision which may change in the future. It seemed most sensible to compartmentalise the database rather than have every function stored in a single table - that is why there are multiple tables all with the same structure

The table structure is very simple, consisting of four columns:

- Name
- Code
- Description
- Storage timestamp

```
     ∇ CodeNewTable tabname;rc;UML
      ⍝ Create a new table for code
       UML←3
       rc←SQAConnect'c1'CodeBank
       rc←SQADo'c1'('create table ',tabname,' (fooname text(50), foocode memo,
foodesc text(50), foostamp datetime)')
       rc←SQAClose'c1'
     ∇
```

```
CodeNewTable=: 3 : 0
NB. Create a new table for code storage
ch=.ddcon 'dsn=codebank'
t=.'create table ',y.,' (fooname text(50), foocode memo, foodesc text(50),foostamp
datetime)'
t ddsql ch
dddis ch
)
```

CodeTableList

List all the tables in the database; this is rather clunky, because of problems experienced with the particular ODBC driver used.

```
     ∇ z←CodeTableList;rc;UIO;UML
      ⍝ List of tables in the code bank
       UML←3
       UIO←0
       z←SQAConnect'c1'CodeBank
       (rc z)←SQATables'c1'
       rc←SQAClose'c1'
       z←1↓[0]z
```

```
[8]     z←((⊂'TABLE')≡¨z[;3])⌿z[;2]
      ∇
```

```
CodeTableList=: 3 : 0
NB. List of code tables
NB. Due to ODBC driver limitations this has to be done using DDE
NB. Assumes that Access is already running
z=.wddata 'ddereq msaccess codebank tablelist;'
}:&.><;.2 z,9{a.
)
```

CodeDropTable

We can easily drop code tables.

```
      ∇ CodeDropTable tabname;rc;UML
[1]    ⍝ Drop a code table
[2]     UML←3
[3]     rc←SQAConnect'c1'CodeBank
[4]     rc←SQADo'c1'('drop table ',tabname)
[5]     rc←SQAClose'c1'
      ∇
```

```
CodeDropTable=: 3 : 0
NB. Drop a code storage table
ch=.ddcon 'dsn=codebank'
('drop table ',y.) ddsql ch
dddis ch
)
```

CodePut

At this point we take advantage of the opportunity to include a description of the function which is quite distinct from the body of the function.

```
      ∇ ∆table CodePut ∆foodesc;vr;desc;stamp;rc;cur;bind;∆foo;∆desc;UML
[1]    ⍝ Put new function onto the table
[2]     UML←3
[3]     →((2≠⍴∆foodesc)∨2≠≡∆foodesc) Err
[4]     (∆foo ∆desc)←∆foodesc
[5]     vr←UVR ∆foo
[6]     stamp←,'G<9999-99-99 99:99:99>'UFMT 100⊥6 UTS
[7]     rc←SQAConnect'c1'CodeBank
[8]     bind←'(:a<C',(⍕⍴∆foo),':,:b<C',(⍕⍴vr),'(L):,:c<C',(⍕⍴∆desc),':,:d<C19(S):)'
[9]     rc←SQADo'c1'('insert into ',∆table,' values ',bind)∆foo vr ∆desc stamp
[10]    rc←SQAClose'c1'
[11]    →0
[12]   Err:'CODEBANK ERROR'USIGNAL 502
      ∇
```

```
CodePut=: 3 : 0
NB. Store a verb definition
NB. This needs a quote-doubler to handle character strings in definitions
:
ch=.ddcon 'dsn=codebank'
N=.'''',(>0{y.),''','
C=.'''',(scriptform >0{y.),''','
D=.'''',(>1{y.),''','
S=.'''',(StampOut 6!:0 ''),''''
t=.'insert into ',(x.),' (fooname,foocode,foodesc,foostamp) values (',N,C,D,S,')'
t ddsql ch
dddis ch
)
```

CodePutAll

Bulk additions, useful for initial table loading, taking advantage of a convention that all functions begin with an overall descriptive comment.

```
      ∇ ∆table CodePutAll ∆foolist;vr;desc;stamp;rc;cur;bind;∆foo;∆desc;UML;i;UIO
[1]    ⍝ Put set of functions onto the table
[2]     UML←3
[3]     UIO←1
[4]     rc←SQAConnect'c1'CodeBank
```

```
[5]    i←0
[6]   Next:→((i←i+1)>ρ∆foolist) End
[7]    ∆foo←i⊃∆foolist
[8]    vr←⎕VR ∆foo
[9]    ∆desc←(vr≠⎕TC[2])⊂vr
[10]   ∆desc←∊(<\'⍝'∊¨∆desc)/∆desc
[11]   ∆desc←(∆desc⍳'⍝')↓∆desc
[12]   stamp←,'G<9999-99-99 99:99:99>'⎕FMT 100⊥6 ⎕TS
[13]   bind←'(:a<C',(⍕ρ∆foo),':,:b<C',(⍕ρvr),'(L):,:c<C',(⍕ρ∆desc),':,:d<C19(S):)'
[14]   rc←SQADo'c1'('insert into ',∆table,' values ',bind)∆foo vr ∆desc stamp
[15]   →Next
[16]  End:rc←SQAClose'c1'
[17]   →0
[18]  Err:'CODEBANK ERROR'⎕SIGNAL 502
     ∇
```

```
CodePutAll=: 3 : 0
NB. Left as an exercise for the reader
)
```

CodeReplace

Here we begin to reap some benefit; the result is a summary of incremental changes between the current and previously-saved versions. The comparison code used is that of Gregg Taylor [Tay94]; *CrFromVr* performs the obvious conversion.

```
     ∇ z←∆table CodeReplace ∆foo;vr;stamp;rc;cur;bind;⎕ML;⎕IO;cursor;data;oldvr
[1]   ⍝ Put new function onto the table
[2]    ⎕IO←0
[3]    ⎕ML←3
[4]    vr←⎕VR ∆foo
[5]    stamp←,'G<9999-99-99 99:99:99>'⎕FMT 100⊥6 ⎕TS
[6]    rc←SQAConnect'c1'CodeBank
[7]    data←SQADo'c1'('select foocode from ',∆table,' where fooname =
:a<C',(⍕ρ∆foo),':')(∆foo)
[8]    →(4≠ρdata) Err
[9]    oldvr←∊ 2⊃data
[10]   oldvr←(-+/∧\⎕AV[4]=⌽oldvr)↓oldvr
[11]   z←25 79 CompFn CrFromVr¨oldvr vr
[12]   z←⊃'*** Previous ***' '*** Current ***'Over¨' ',¨z
[13]   bind←(':a<C',(⍕ρvr),'(L):')(':b<C19(S):')(':c<C',(⍕ρ∆foo),':')
[14]   cursor←'update ',∆table,' set foocode =',(0⊃bind),', foostamp = ',(1⊃bind),'
where fooname =',(2⊃bind)
[15]   rc←SQADo'c1'cursor vr stamp ∆foo
[16]   rc←SQAClose'c1'
[17]   →0
[18]  Err:'CODEBANK ERROR'⎕SIGNAL 504
     ∇
```

```
CodeReplace=: 3 : 0
NB. Replace a verb definition
:
ch=.ddcon 'dsn=codebank'
C=.'''',(scriptform y.),''''
S=.'''',(StampOut 6!:0 ''),''''
t=.'update ',x.,' set foocode = ',C,', foostamp = ',S,' where fooname = ''',y.,''''
t ddsql ch
dddis ch
)
```

CodeGet

Clearly, once the functions are stored we need to be able to retrieve them; and bulk retrieval is more relevant than single retrieval - most of the time. Since the function returns a list of what it got we can easily arrange to have the contents of the list removed at a later date.

```
     ∇ z←∆table CodeGet ∆foospace;rc;cur;data;nul;⎕IO;vr;⎕ML;i;∆foo;∆space
[1]   ⍝ Retrieve function from table and fix in workspace
[2]    ⎕IO←0
[3]    ⎕ML←3
[4]    'CODEBANK ERROR'⎕SIGNAL(2≠ρ∆foospace)/501
[5]    (∆foo ∆space)←∆foospace
[6]    ⍎(1=≡∆foo)/'∆foo←,⊂∆foo'
[7]    rc←SQAConnect'c1'CodeBank
[8]    z←''
[9]    i←¯1
[10]  Next:→((i←i+1)=ρ∆foo) End
[11]   data←SQADo'c1'('select foocode from ',∆table,' where fooname =
:a<C',(⍕ρi⊃∆foo),':')(i⊃∆foo)
[12]   →(4≠ρdata) Err
```

```
  vr←∊ 2⊃data
  vr←(+/∧\vr=UAV[4])↓(-+/∧\UAV[4]=⌽vr)↓vr
  ⍎'z←z,⊂',∆space,'.UFX vr'
  →Next
 End:rc←SQAClose'c1'
  →0
 Err:rc←SQAClose'c1'
  'CODEBANK ERROR'USIGNAL 501
∇
```

```
CodeGet=: 3 : 0
NB. Retrieve and fix a verb definition
:
ch=.ddcon 'dsn=codebank'
i=._1
while. (i=.i+1)<$y. do.
    sh=.('select foocode from ',x.,' where fooname = ''',(>i{y.),'''') ddsel ch
    z=.ddfet sh
    0 !: 100 >z
end.
dddis ch
)
```

CodeErase

With all good things coming to an end we may want to erase code from a table...

```
∇ ∆table CodeErase ∆foo;rc
 ⍝ Delete function from table
  rc←SQAConnect'c1'CodeBank
  rc←SQADo'c1'('Delete from ',∆table,' where fooname = :a<C',(⍕⍴∆foo),':')∆foo
  rc←SQAClose'c1'
∇
```

```
CodeErase=: 3 : 0
NB. Delete a verb definition from a table
:
ch=. ddcon 'dsn=codebank'
t=.'delete from ',x.,' where fooname = ''',y.,''''
t ddsql ch
dddis ch
)
```

CodeCatalogue

An obvious requirement, and the first step towards analysing and rationalising what we have.

```
∇ z←{sortcol}CodeCatalogue ∆table;data;UIO;rc;UML;sortcols
 ⍝ Catalogue of functions in table
  UML←3
  UIO←0
  ⍎(0=UNC'sortcol')/'sortcol←''name'''
  sortcols←'name' 'date'
  sortcol←0 2 0[sortcols⍳⊂sortcol]
  rc←SQAConnect'c1'CodeBank
  →(4≠⍴data←SQADo'c1'('select fooname, foodesc, foostamp from ',∆table)) Err
  rc←SQAClose'c1'
  z←Dtb¨ 2⊃data
  z←z[UAV⍋⊃z[;sortcol];]
  z←'Name' 'Description' 'Stamp',[0]' ',[0]z
  →0
 Err:rc←SQAClose'c1'
  'CODEBANK ERROR'USIGNAL 503
∇
```

```
CodeCatalogue=: 3 : 0
NB. Formatted cataloge of verb definitions in table
ch=.ddcon 'dsn=codebank'
sh=.('select fooname,foodesc,foostamp from ',y.) ddsel ch
res=.ddfet sh, _1
ddend sh
dddis ch
res=.(<Dtb >0{"1 res),"1 (<Dtb > 1{"1 res),(<19{."1> 2{"1 res)
((<'name'),(<'desc'),(<'stamp')),:res
)
```

CodeList

This short form catalogue clearly helps us to get everything, and to move on toward having logical groupings within tables.

```
     ∇ z←CodeList ∆table;data;UIO;rc;UML
      ⍝ List of functions in table
       UML←3
       UIO←0
       rc←SQAConnect'c1'CodeBank
       →(4≠⍴data←SQADo'c1'('select fooname from ',∆table)) Err
       rc←SQAClose'c1'
       z← 2⊃data
       z←Dtb¨,z[UAV⍋⍉z;]
       →0
     Err:rc←SQAClose'c1'
       'CODEBANK ERROR'USIGNAL 505
     ∇
```

```
CodeList=: 3 : 0
NB. List of verb definitions held in table
ch=.ddcon 'dsn=codebank'
sh=.('select fooname from ',y.) ddsel ch
res=.ddfet sh, _1
ddend sh
dddis ch
res=.,"2 >res
(<"1 res) Without&.> ' '
)
```

CodeSearch

Now we can go hunting for functions which are there, somewhere.

```
      Uvr 'CodeSearch'
     ∇ z←CodeSearch foo;tables;list
      ⍝ Tables which contain the function
       tables←CodeTableList
       z←((⊂⊂foo)∊¨CodeList¨tables)/tables
     ∇
```

```
CodeSearch=: 3 : 0
NB. List of tables containing verb definition
tables=.CodeTableList ''
lists=.CodeList&.>tables
(>(<<y.) e.&.>lists)#tables
)
```

Future Extensions

Given the above starting set we can contemplate the future:

Function code is stored outside the originating APL environment, which means that it is accessible from any other APL used on the hardware; given the ODBC capabilities of APL*PLUS III, for example, we could implement this same set of functions. All that remains is to add in atomic vector translation and we have shared access to a common pool of defined functions from both interpreters. The same could hold true for APL2. Of course, we are still at the mercy of dialectical independence - but one barrier is down.

As you can see, there is nothing here which makes use of Access itself - ODBC has done its job and insulated us from the characteristics of the underlying database. All that APL has to do is to line up the SQL. We can use alternative database systems on different architectures.

One obvious future step is to store the `CodeReplace` results on the database; we could keep a complete set of historic code versions, but practical experience with function file systems which have this capability is that it is too rarely used.

As Baker points out, having a description separate from the code is the first step toward storing a number of relevant items of code documentation alongside (but not embedded within) the code itself. We can automate enforcement of documentation standards.

Function editing (and auto-saves) can be integrated into the procedure.

Cross-references and dependencies may be analysed, and indeed stored as part of the overall dictionary system.

Summary

The paper has described a workable initiation of an APL Dictionary Database System; single copies of functions may be stored and shared across interpreters. The programmer actively puts code where it should be stored.

Off loading the mechanics of data storage onto an independent database system relieves the APL programmer of much of the tedium involved in building similar function filing systems wholly in APL.

The mechanism is not dependant on any single database product.

Performance appears to be adequate for applications in which function retrieval is an operation which occurs relatively infrequently (the overheads of ODBC would probably preclude use to fetch single functions in dynamic response to *VALUE ERROR*, without the user noticing).

References

[Bak94] Using FoxPro and DDE to Store J Words; John D Baker; APL94 Conference Proceedings Companion

[Bow94] New Graphical User Interface Proposal for APL; Dick Bowman; APL Quote Quad Volume 25 Number 1 pp 17-22.

[Tay94] Extension Notes; Gregg Taylor; The APL Perspective July 1994 pp 19-22.

Code Notes

This version of the code is offered into the public domain; it may be copied or modified in any way. The only restriction on distribution is that you must acknowledge Dogon Research as the originator.

The code is written using Dyalog APL/W Version 7.1 and J Version 2.03; neither Dogon Research, Iverson Software or Dyadic Systems accept any responsibility for malfunction or incompatibility.

The version of Microsoft Access used is Release 1.1, and the ODBC driver is that supplied as part of the Iverson Software J2 Release 2.03. All trademarks are acknowledged.

As you will see, there is no attempt to make the code especially robust; note that some of the lines are wrapped around. Note also that some problems were experienced with the ODBC driver syntax, the statements shown here 'work' without necessarily being the most direct expression of intent.

The J code uses a number of ustilities which are included here for completeness; as the author is quite new to J the code here is almost certainly less than ideal.

```
Dtb=: 3 : 0
NB. Delete trailing blanks (there must be a shorter way)
(0 ,- +//*./\ |.*./ ' '=y.)}. y.
)


StampOut=: 3 : 0
NB. Format timestamp for the database (probably a driver limitation)
z=.4 3 3 3 3 3":y.
z=.'-' 4 7}z
z=.':' 13 16}z
z=.'0' ((z=' ')#i.$z)}z
z=.' ' 10}z
z
)


Without=: 3 : 0
NB. Exclude characters
:
   (-.y.=x.)#x.
   )
```

Return to the J\APL Home Page