CMS Pipelines Tips and Techniques

Martin Zimelis

Ross Patterson

Computer Associates, Int'l Inc.

Agenda

- Part I - How *Not* to Program in CMS Pipelines
- Part II - Looping and Calculating in pipes
- Part III - Flowing Text
- Part IV - Integrating User Stages into Pipeline Execs

Part I - How *Not* to Program in CMS Pipelines

- CMS Pipelines is an extremely powerful productivity enhancer, but easy to misuse
- Many novice plumbers pick up enough knowledge of Pipelines programming techniques to use it badly
- Most presentations address the “how to” aspect of Pipelines programming
- We address some of the abuses of Pipelines programming techniques and how to avoid them by presenting a series of examples from actual code

“Not thinking like a plumber”

```
‘PIPE < data file | stem recs.’
Do i = 1 To recs.0
  :
  :
End i
‘PIPE stem newrecs. | > data file a’
```

- If you can’t do more with Pipelines than fill a stem or write to a file, use EXECIO
- Less bad: Select or transform records before putting them in the stem
- Best: Do the loop processing inside the pipeline

Multistream Pipes Aren't *That* Scary

- The problem:

  Read a file. Identify one type of record to be deleted from the file. Put that type of record in a log file. Process the remaining records.

Multistream Pipes Aren't *That* Scary

- **The problem:**

  Read a file. Identify one type of record to be deleted from the file. Put that type of record in a log file. Process the remaining records.

```
'PIPE < data file a | stem recs.'
```

(identify the records to be deleted)

```
'PIPE stem recs. | locate 1.8 /'key'/ | >> deleted log d'
'PIPE stem recs. | nlocate 1.8 /'key'/ | stem recs.'
```

(process the records)

Multistream Pipes Aren't *That* Scary

- The problem:

  Read a file. Identify one type of record to be deleted from the file. Put that type of record in a log file. Process the remaining records.

```
'PIPE < data file a | stem recs.'
```

(identify the records to be deleted)

```
'PIPE stem recs. | locate 1.8 /'key'/ | >> deleted log d'
'PIPE stem recs. | nlocate 1.8 /'key'/ | stem recs.'
```

(process the records)

- For the last part, try the following instead:

```
'PIPE (end ?) stem recs. | n: nlocate 1.8 /'key'/ | stem
   recs.',
      '? n: | >> deleted log d'
```

It runs 35% faster on 22000 80-byte records.

Even LOOKUP Isn’t That Scary…

- Not using “Pipethink”

```
del_codes = ’SQ SD SM SL SKX SKV SRD SRE SKT SKB SKI SKC SRN SRS’
num_codes = words(del_codes)
do i = 1 to num_codes
  ’Pipe < HRIMSCHG FILE * |’,
        ’locate 111-115 /’left(word(del_codes,i),5)’/ |’,
      'specs 32-40 1 111-115 11 |’,
      'stem delids. append '
end
Return
```

- Pipethink rule #2: If you’re coding a pipeline inside a loop, you’re not using Pipethink

…Even LOOKUP Isn't That Scary

- Using the right stage

```
 del_codes = 'SQ SD SM SL SKX SKV SRD SRE SKT SKB SKI SKC SRN SRS'
'PIPE (end ?)',
     '< HRIMSCHG FILE |',
    'L: lookup pad blank 111.5 1.5 detail |',
     'specs 32-40 1 111-115 11 |',
     'stem delids.',
   '?',
     'var del_codes |',
     'split |',
    'L:'
```

- Think about the problem like a plumber.
  - When looking for multiple values in the same field, think **LOOKUP**, not loop

The Same Problem, But “Unfolded”

```
’PIPE < XACTSND REQUEST A | STEM ALLREQ.’
 trec. = ’’;trec.0 = 0
 ’PIPE STEM ALLREQ.  | LOCATE 1-7 /EXCHGAD/’,
              ’ | STEM TREC.’
  call XACTLOG ’XACTSND there are ’trec.0’ EXCHANGE adds’

  temp. = ’’;temp.0 = 0
 ’PIPE STEM ALLREQ.  | LOCATE 1-7 /EXCHGDE/’,
              ’ | STEM TEMP. ’
  call XACTLOG ’XACTSND there are ’temp.0’ EXCHANGE dels’
  if temp.0 > 0 then ’PIPE STEM TEMP. | stem trec. append’

  temp. = ’’;temp.0 = 0
 ’PIPE STEM ALLREQ.  | LOCATE 1-7 /EXREA  /’,
              ’ | STEM TEMP. ’
  call XACTLOG ’XACTSND there are ’temp.0’ EXCHANGE reactivates’
  if temp.0 > 0 then ’PIPE STEM TEMP. | stem trec. append’

  temp. = ’’;temp.0 = 0
 ’PIPE STEM ALLREQ.  | LOCATE 1-7 /NTADD  /’,
          :
          :  (for a total of 10 different keys)
```

How One Might Do It

```
xacttype = ’EXCHGAD EXCHANGE adds;EXCHGDE EXCHANGE dels;’,
         ||’EXREA EXCHANGE reactivates;NTADD NT Only adds;’,
         ||’NTADDE NT Only Extended adds;NTCHGE NT Only Extended changes;’,
         ||’NTDELNT Only dels;NTADX NT ADDX;’,
         ||’EXCHGAX EXCHANGE ADDX’
            NTDEL NTADX EXCHGAX’
’PIPE (end ?)’,
   ’stem allreq. |’,
  ’L: lookup 1.7 count detail |’,
   ’stem trec.’,
 ’?’,
   ’var xacttype |’,
   ’split ; |’,
  ’L:’,
 ’? L:’,
 ’? L: |’,
  ’spec /EXEC XACTLOG "XACTSND there are/ 1.10 strip nw 18-* nw /"/ n |
  ’command’
```

- Getting over “multi-stream phobia” opens many doors

Sometimes Pipelines is the Wrong Tool

- To wit

```
 out = Date('S') Time() Left('UPDATE',8) Left(Userid(),8)
'PIPE var out | >> update log d'
```

- Use instead

```
'EXECIO 1 DISKW UPDATE LOG D 0 (STRING',
   Date('S') Time() Left('UPDATE',8) Left(Userid(),8)
```

Part II - Looping and Calculating in pipes

- Two examples presented
- Both are the result of tool development
- Both use 407 emulation
- The first one includes a looping pipe

Tool for VM: Webgateway

- Background
  - CGI parameters are presented one of two ways, depending on how program was called
    - METHOD="GET" - via HTTP environment variable QUERY_STRING

      'CGI GETVAR QUERY_STRING (VAR QS'
    - METHOD="POST" - via the input stack

      'CGI READ 1 (VAR QS TRANSLATE USENGLISH'
  - In each case, the value returned is the "URL-encoded" form of the parameters

    URL**?NAME1=VALUE1&NAME2=VALUE2&** . . .

VM:Webgateway Background

- The **CGI URLDECODE** command
  - performs translation of encoded characters
  - splits the parameters into individual variables

    ‘CGI URLDECODE (VAR QS INTO PARMS.’
  - The PARMS.0 variable contains the list of parameter names

    ```
      Say parms.0
    NAME1  NAME2   . . .
    ```

    which leads to

    ```
      Say parms.name1
    VALUE1
      Say parms.name2
    VALUE2
    ```

Where Is He Going With This?

- A customer gave us a WIBNI

  If I don't need to know, for any other reason, how the CGI program was invoked, wouldn't it be nice if there were just one way to retrieve its parameters.

- So we wrote a little EXEC

  - It encapsulates this idea and materializes the result in its caller's name space

    ```
    Call GetParms
    ```

  would produce PARMS.0 and all the little PARMS..

The Basic Logic

```
‘pipe’,
   ‘var parms.0 |’,         /* Variable names  */
   ‘split |’,               /* One per record  */
   ‘strip |’,               /* Cleanliness     */
   ‘literal PARMS.0|’,      /* Include main    */
   ‘spec /(stagesep !) var/ 1’,
         ‘w1 nw’,           /* Build a pipe to */
         ‘/! parms./ nw’,   /* promote to next */
         ‘w1 n’,            /* level           */
         ‘/ 1/ n |’,
   ‘runpipe’                /* Run it          */
```

- This works fine until you get a **SELECT MULTIPLE** tag or something else that produces duplicate names

A Bit of Complexity

- If a user selects two (or more) items from a **SELECT** box with the **MULTIPLE** option specified

  URL?**NAME1=VALUE1&NAME2=VALUE2A&NAME2=VALUE2B** . . .

- After **URLDECODE…**

```
   Say parms.0
 NAME1   NAME2   NAME2 . . .
   Say parms.name2
 VALUE2B
```

- What we don't see from **PARMS.0** is that a sub-stem has also been created

PARMS.NAME2.0	**PARMS.NAME2.1**	**PARMS.NAME2.2**
2	VALUE2A	VALUE2B

The Problem

- Construct the complete set of variables created by URLDECODE and materialize them into GetParms caller's name space
- The conceptual solution
  - Determine how many occurrences of each name appear in **PARMS.0**
  - For those that appear once, process as before
  - For those that appear more than once
    - pass along one copy of the name for processing as before
    - construct a stem with as many entries as there are occurrences of the name; don't forget the count entry (**..0**)

The Solution Details

- Identify the “multiples”

```
  ‘pipe’,
      ‘var parms.0 |’,
      ‘split |’,
      ‘strip |’,
      ‘sort |’,
      ‘unique count last |’,
```

- Now we have records of the form

```
__________2NAME2
   10 chars
```

Passing the Original Variables

- Since all “simple stem” variables have to be passed, make a copy of the records

```
   ‘o: fanout |’,
```

- From one of those streams, select just the multiples

```
     ‘pick 1.10 >> /          1/ |’,
```

- Use the count and the name to construct an element of the second level stem

```
     ‘spec 11-* 11’,
           ‘/./ n’,
           ‘1-10 strip n’,
```

Generating the Substem Elements

- We still need to generate the remaining substem elements, including the count
- Using 407 emulation to do some arithmetic

```
'spec 11-* 11',
      '/./ n',
     'c:1-10 strip n write',
      'print c-1 1.10 right',
      '11-* n |',   /* Rec w/ decremented count */
```

- To make it really spiffy, route the two types of records to different output streams

```
's: spec 11-* 11',
        '/./ n',
       'c:1-10 strip n write',
        'outstream 1',
        'print c-1 1.10 right',
        '11-* n |',
```

Creating a Loop

- In order to generate all of the elements of the sub-stem, route the secondary output of the **specs** stage back to its primary input
  - This is done by putting an **elastic** stage immediately prior to the **specs** stage

```
 ‘pick 1.10 >> /     1/ |’,
 ‘e: elastic |’,
 ‘s: specs 11-* 11’,
         :
‘?’,
 ‘s: |’,
 ‘e:’
```

- We still need a way to terminate the loop
  - As written, it would decrement the count forever

“Closing” the Loop

- Terminate the loop when the count goes negative
  - When a ‘-’ appears in the count field (columns 1-10)

```
         :
         :
   ‘s: |’,
     'nlocate 1-10 /-/ |’,
    ‘e:’
```

Putting the Pieces Together

- Gather the variable names
  - The secondary output of **fanout**
  - The primary output of **specs**
- Generate small pipe specifications to export the variables to the caller’s name space
  - After the first **specs** stage…

```
 ‘i: faninany |’,
  ‘spec /(stagesep !) var parms./ 1
       ‘11-* n’,
       ‘/ ! var parms./ n’
       ‘11-* n / 1/ n |’,
  ‘runpipe’,
‘?’,
 ‘o: | i:’
```

The Complete Solution (pt1)

```
’pipe (end ? name GETPARMS)’,
   ’var parms.0 |’,                 /* Take the list of variables    */
   ’split |’,                       /* Split it to one per record    */
   ’strip |’,                       /* Left justify them             */
   ’literal PARMS.0|’,              /* Include the .0 variable       */
   ’sort |’,                        /* Set up to look for dups       */
   ’unique count last |’,           /* Identify how many of each     */
  ’o: fanout |’,                    /* Keep a copy of the original   */
   ’pick 1-10 >> /         1/ |’,   /* Only interested in multiples  */
  ’e: elastic  |’,                  /*   (Loop stall control)        */
  ’s: spec’,                        /* Build two records:            */
        ’11-* 11’,                  /*  First is one element of      */
        ’/./ n’,                    /*   the multi-level stem        */
       ’c: 1-10 strip n write’,     /*   (e.g., PARMS.B.2).          */
        ’outstream 1’,              /*  Second is like the original  */
        ’print c-1 1.10 right’,     /*   record with the count       */
        ’11-* n |’,                 /*   decremented by one.         */
  ’i: faninany |’,
```

The Complete Solution (pt2)

```
  ’spec’,                          /* Prefix each var name with    */
     ’/(stagesep !) var parms./ 1’,   /* the stem root.  Construct */
     ’11-* n’,                     /*  a pipe to pass the local    */
     ’/ ! var’ parms.’/ n’,        /*  copy of each variable up to */
     ’11-* n’,                     /*  the caller.                 */
     ’/ 1/ n |’,
  ’runpipe’,
’?’,
 ’s: |’,                           /* If the count remains non-    */
  ’nlocate 1.10 /-/ |’,            /*  negative, route the record  */
 ’e:’,                             /*  back for another pass.      */
’?’,
 ’o: |’,                           /* Send a copy of the original  */
 ’i:’                              /*  variable back to the caller */
```

Another 407 Emulation Example

- Working with counters
  - CPMonitor data (for example) accumulates in counters that do not reset between system IPLs.
  - They are perpetually increasing (decreasing) in value
  - To get the count of something between two observations, compute the difference between the observations

	Counter	Difference
Obs 1:	14837	?
Obs 2:	15112	275
Obs 3:	15399	287

Differencing Counters - Method 1

- One method of differencing takes advantage of the 407’s two reading stations

```
/* Example of using 407 emulation to do field differencing */
   ’Callpipe (name DIFF0)’,
      ’*: |’,
      ’spec w1-3 1’,                         /* Just pass along other fields */
          ’s:w4 .’,                          /* Specify an identifier        */
           ’select second’,                  /* Specify previous record      */
          ’f:w4 .’,                          /* Identifier on the prior rec  */
           ’print s-f picture sssssssss9 nw |’,   /* Output difference */
      ’drop 1 |’,                            /* First one isn’t meaningful   */
      ’*:’
```

- Just one shortcoming exists: use of the second reading station delays the record

Differencing Counters - Method 2

- An alternative approach exists, using two counters and one reading station

```
/* Example of using 407 emulation to do field differencing */
   ’callpipe (name DIFF1)’,
      ’*: |’,
      ’spec w1-3 1’,                   /* Just pass along other fields */
          ’s:w4 .’,                    /* Specify an identifier        */
           ’set #1:=s-#0’,             /* Compute difference           */
           ’set #0:=s’,                /* Save "previous" value        */
           ’print #1 picture sssssssss9 nw |’,    /* Output it         */
      ’drop 1 |’,                      /* First one isn’t meaningful   */
      ’*:’
```

- This approach relies on the fact that counters are initialized to 0
- In both cases, for decrementing counters, simply reverse the order of arguments in the subtraction

Part III - Flowing Text
A more elegant approach

- The text flowing problem has been around for many years
  - Flow blocks of text together making lines that are less than or equal in length to a specified value while preserving paragraph breaks.
- Marty had a Pipeline solution for a number of years but “I wasn’t happy with it”
  - It used a subroutine Pipeline in a loop
  - It was slow
- Attempts to solve the problem without a Callpipe for each “paragraph” met with repeated failures

The Problem

- A paragraph is defined as a block of lines that
  - is delimited by blank lines
  - begins with an indented line (one or more leading blanks)
  - or both
- Marty couldn't find a solution that correctly dealt with all three cases
  - Preceding blank line, first line indented
  - Preceding blank line, first line not indented
  - No preceding blank line, first line indented (the "obvious" fourth case isn't the start of a paragraph)

The Three Cases - Pictorially

```
...~~~~ ~~ ~~~~~~~~ ~~~~
~~~~~.

   ~~~ ~~~~~ ~~~~~ ~~~
~~~~~~ ~~~~ ~~~ ~~~~
   ~~~,
~~~ ~~ ~~ ~~~~~ ~~~~~~~~
~~~~.

   ~~~~ ~~ ~~~ ~~~~~~~
~~~~~~~~~...
```

```
...~~~~ ~~ ~~~~~~~~
   ~~~~
~~~~~.
   ~~~ ~~~~~ ~~~~~ ~~~
~~~~~~ ~~~~ ~~~ ~~~~
   ~~~,
~~~ ~~ ~~ ~~~~~
   ~~~~~~~~~
~~~~.
   ~~~~ ~~ ~~~ ~~~~~~~
~~~~~~~~~...
```

```
...~~~~ ~~ ~~~~~~~~ ~~~~
~~~~~.

~~~ ~~~~~ ~~~~~ ~~~
   ~~~~~~
~~~~ ~~~ ~~~~ ~~~, ~~~
   ~~
~~ ~~~~~ ~~~~~~~~~~
   ~~~~.

~~~~ ~~ ~~~ ~~~~~~~
~~~~~~~~~...
```

The Environment

- The solution would appear in an environment such as this:

```
‘pipe’,
   ‘< input file |’, /* The data source */
   ‘textflow 72 |’,  /* Lines <=72 chars*/
   ‘> output file’   /* The data sink   */
```

The “Old” Solution

```
/* TextFLOW records, breaking at any line that begins with a blank
   (including empty lines)
*/
   Parse Arg width
   Do Forever
      ’peekto line’                 /* Ensure there’s something there */
      If Rc ¬= 0 Then Exit (Rc¬=12)*Rc
      If Strip(line) = '' Then Do   /* Just pass any empty lines */
         'output' line
         'readto'
      End
      Else Do
         'callpipe (name TFLOW)',
            '*: |',
            'drop 1 |',           /* Don't look at first line (might   */
                      ,           /*                begin with a blank) */
            'pad 1 |',            /* Ensure at least 1 blank */
            'strtolabel / /|',    /* Pass records until the next break */
            'literal' line'|',    /* Replace first line of group */
            'join * / / |',
            'spill' width '|',    /* Flow to width */
            '*:'                  /* Pass to caller */
      End
   End
```

A Flash of Insight

- While reviewing the latest version of Melinda Varian's "Streamlining Your Pipelines," Marty had an epiphany
- There probably was no all-encompassing single stream solution to the problem
- The insight was that, in order to reconstruct paragraphs after flowing, I had to identify their boundaries

(Yes, I know. Obvious once you think of it.)

The New Solution

```
/* FLOW REXX:
   Flow each paragraph in a stream of text to the width specified
   in the argument.
*/
   Parse Arg width .
   ’callpipe (end ? Name FLOW)’,
      ’*: |’,
     ’l: locate w1 |’,                 /* Isolate blanks lines         */
     ’n: nfind _|’,             /* Lines w/ leading blank also delimit */
     ’i: faninany |’,
      ’joincont not leading x00 / / |’,   /* One record per paragraph  */
      ’change x0140 x01 |’,            /* See block comment below      */
      ’split x01 |’,                   /* Formerly blank line restored */
      ’spill’ width ’|’,               /* Spill to desired width       */
      ’change x00 // |’,               /* Remove paragraph markers     */
      ’pad 1 |’,                       /* Retain blank lines if        */
      ’*:’,                            /*  writing to a file.          */
    ’?’,
     ’l: |’,
      ’spec x0001 1 |’,                /* Replace empty line with 00 01*/
     ’i:’,
    ’?’,
     ’n: |’,
      ’change // x00 |’,               /* Prepend marker character     */
     ’i:’
```

The New Solution (continued)

```
/*
   A paragraph is, for purposes of this stage, a block of text that
   is delimited by a blank line or by a first line that is indented
   one or more spaces (or both).

   The first CHANGE stage (CHANGE X0140 X01) handles the case where
   paragraphs are delimited by blank lines and are NOT indented.  In
   that case, the JOINCONT stage will join the ’0001’x blank line
   marker with the line of text that follows, inserting a single
   blank between them.  This blank must be removed.
*/
```

The Acid Test -- Performance

- The obvious questions is “How do they compare in resource consumption?”
- We chose one of Marty’s NOTEBOOK files as an input source
  - 1148 records (format V 181)
  - almost 48K bytes
- Lovely Rita gave us the answer

Performance -- Before

```
CPU Utilization by Pipeline Specification      from: 12 Jan 1999 09:22:54
                                                 to: 12 Jan 1999 09:23:24

CPU utilization of pipeline specification "NoName001":
     5.980 (     5.980) ms (  53K) in stage   1 of pipeline  1: < share notebook
   148.040 (   104.104) ms (   3K) in stage   2 of pipeline  1: tflow2 72
    11.034 (    11.034) ms (  65K) in stage   3 of pipeline  1: > eraseme file a
   165.054 (   121.118) ms total in "NoName001" (1 invocation) <=====

CPU utilization of pipeline specification "TFLOW":
     4.141 (     4.141) ms (  <1K) in stage   2 of pipeline  1: drop 1
     6.455 (     6.455) ms (  <1K) in stage   3 of pipeline  1: pad 1
     9.447 (     9.447) ms (  <1K) in stage   4 of pipeline  1: strtolabel / /
     1.682 (     1.682) ms (  <1K) in stage   5 of pipeline  1: literal ========
    13.917 (    13.917) ms (   7K) in stage   6 of pipeline  1: join * / /
     8.294 (     8.294) ms (   1K) in stage   7 of pipeline  1: spill 72
    43.936 (    43.936) ms total in "TFLOW" (243 invocations) <=====
```

Performance -- Before (continued)

```
165.054 ms attributed to stages; no virtual I/O.
    2 pipeline specifications used (9 stages).
  244 pipeline specifications issued.

    0.023 ms in general overhead.
  189.402 ms in scanner.
   14.625 ms in commands.
  142.823 ms in dispatcher.
  212.808 ms in accounting overhead.

  724.735 ms total.
```

Performance -- After

```
CPU Utilization by Pipeline Specification      from: 11 Jan 1999 15:28:39
                                                 to: 11 Jan 1999 15:28:40

CPU utilization of pipeline specification "FLOW":
    11.783 (    11.783) ms (   1K) in stage   2 of pipeline  1: locate w1
     6.105 (     6.105) ms (  <1K) in stage   3 of pipeline  1: nfind _
     6.601 (     6.601) ms (  <1K) in stage   4 of pipeline  1: faninany
    15.880 (    15.880) ms (   6K) in stage   5 of pipeline  1: joincont not
      leading x00 / /
     7.845 (     7.845) ms (   7K) in stage   6 of pipeline  1: change x0140 x01
     5.570 (     5.570) ms (   1K) in stage   7 of pipeline  1: split x01
     4.230 (     4.230) ms (   1K) in stage   8 of pipeline  1: spill 72
    12.119 (    12.119) ms (   1K) in stage   9 of pipeline  1: change x00 //
     4.758 (     4.758) ms (  <1K) in stage  10 of pipeline  1: pad 1
     4.304 (     4.304) ms (   4K) in stage   2 of pipeline  2: spec x0001 1
     2.326 (     2.326) ms (   1K) in stage   2 of pipeline  3: change // x00
    81.521 (    81.521) ms total in "FLOW" (1 invocation) <=====

CPU utilization of pipeline specification "NoName001":
     3.669 (     3.669) ms (  53K) in stage   1 of pipeline  1: < share notebook
    84.438 (     2.917) ms (   3K) in stage   2 of pipeline  1: flow 72
     9.523 (     9.523) ms (  65K) in stage   3 of pipeline  1: > eraseme file a
    97.630 (    16.109) ms total in "NoName001" (1 invocation) <=====
```

Performance -- After (continued)

```
97.630 ms attributed to stages; no virtual I/O.
     2 pipeline specifications used (14 stages).
     2 pipeline specifications issued.

     0.021 ms in general overhead.
     2.709 ms in scanner.
     0.065 ms in commands.
    75.216 ms in dispatcher.
     2.421 ms in accounting overhead.

   178.062 ms total.
```

- The bottom line: CPU consumption in this case decreases by 75% (1 - 178.062/724.735)
- Not surprising, but very satisfying
- The moral: A subroutine pipeline in a loop can be a very expensive solution

Part IV - Integrating User Stages into Pipeline Execs

- User-Written Stages
  - Are created when
    - built-in stages can't do the task at hand
    - you want to write a "macro" of pipeline stages
  - Typically appear in a file with a filetype of REXX
  - Are called by name
- But there's another way
  - to write them
  - to incorporate them in a pipeline

Another Way to Invoke a User-Written Stage

```
‘Pipe (end ?)’,
              :
              :
  ‘Label: rexx *.1: |’,/*The insertion point*/
              :
              :
  ‘?’,
     ‘var uws |’, /* The user-written stage*/
    ‘Label:’
```

where uws looks like this

```
  uws = “/* Comment */ stmt_1; stmt_2; . . . ”
```

Why Would Anyone Want to Do That?

- Makes application maintenance easier
  - There's only one file to transport
  - No worry of "does the version of the stage match the version of the EXEC?"
- Marginally faster than using an external file for the user-written stage
  - Pipelines doesn't have to find and read another file
- Most effective if the user-written stage is unique to one pipeline

How Do I Get There?

- How do I turn this

```
/* Comment */
Signal On Error
Do Forever
   ‘peekto rec’
        :
        :
   ‘output’ something
   ‘readto’
End
```

- Into this

```
“/* Comment */;”,
“Signal On Error;”,
“Do Forever;”,
   “‘peekto rec’;”,
        :
        :
   “‘output’
    something;”,
   “‘readto’;”,
“End;”
```

conveniently?

Converting a User-Written Stage

- Add punctuation using MKSTAGE EXEC

```
mkstage uws rexx a = insert =
```

  - (MKSTAGE EXEC is included as the last page of this presentation or drop the author a note for a machine-readable copy)

- Edit file containing the parent pipeline
  - Insert the file containing the modified stage *before* the parent pipeline

```
get uws insert a
```

  - Assign the string containing the user-written stage to a variable
  - Modify the parent pipeline to use the now-internal stage according to the technique shown on page 3

The Modified Parent Pipeline

```
‘Pipe’,
  ‘< some file |’,
  ‘locate /Marty/ |’,
  ‘uws |’, /* External
    */
  ‘> modified file a’
```

```
uws = “[contents of
    user-written stage]”

‘Pipe (end ?)’,
  ‘< some file |’,
  ‘locate /Marty/ |’,
 ‘U: rexx (*.1:) |’,
  ‘> modified file a’,
 ‘?’,
  ‘var uws |’,
 ‘U:’
```

```
’PIPE (end ? name MKSTAGE)|’,
   ’<’ infn inft infm ’|’,
   ’change /"/ /""/ |’,           /* Double up any existing ’"’s  */
   ’reverse |’,
   ’strip leading |’,                /* Actually STRIP TRAILING      */
  ’n:nfind , |’,                    /* For non-continuation lines.. */
   ’strip leading ;|’,               /*  Remove any existing ’;’s    */
   ’insert /;/ |’,                  /*  before adding after _every_ */
  ’i1:faninany |’,                  /*  line.                       */
   ’insert /,"/ |’,                 /* Trailing quote and continuat.*/
   ’reverse |’,                     /* Forwards again               */
  ’c: chop 0 before not blank |’,   /* Set up to put the leading    */
  ’j: juxtapose |’,                 /*  quote after any indentation.*/
  ’d: drop last 1 |’,               /* Go remove the trailing comma */
  ’i2: fanin |’,                    /*  on the last line.           */
   ’>’ outfn outft outfm,
 ’?’,
  ’n: |’,                           /* For continuation lines, lose */
   ’strip leading , |’,             /* the now-redundant ’,’        */
  ’i1:’,
 ’?’,
  ’c: |’,                           /* Put leading quote just before*/
   ’insert /"/ |’,                  /*  first non-blank character.  */
  ’j:’,
 ’?’,
  ’d: |’,
   ’strip trailing str /,/ |’,      /* Remove continuation comma on */
  ’i2:’                             /*  last line.                  */
```