Rexxification of Pipelines

2025 - 36th International Rexx Language Symposium

Vienna, Austria and online between May 4 and May 7, 2025

Jeffrey Hennick
Jeff@Jeff-H.com

May 5th, 2025

What and Why

- Rexx and Pipelines were developed at about the same time in different parts of IBM
- Pipelines "language" is terse for easy typing
- Rexx added the concept of "readability" to languages
-
- Recent additions to NetRexx Pipelines add some readability to existing stages, and new stages taking advantage of the power of Rexx, specifically of NetRexx (which also brings the power of Java.)

Three Considerations for Rexxification

1) Existing Stages
2) New Stages
3) Using a pipe in NetRexx

Why and What

- Some existing stages have new, optional, prompt words to aid reading (and writing).
- Some have new NetRexx functionality.

- And some new Rexx-based stages.

SPECS Stage

For example, the SPECS stage has a number of built in conversions and its own "almost Rexx" language

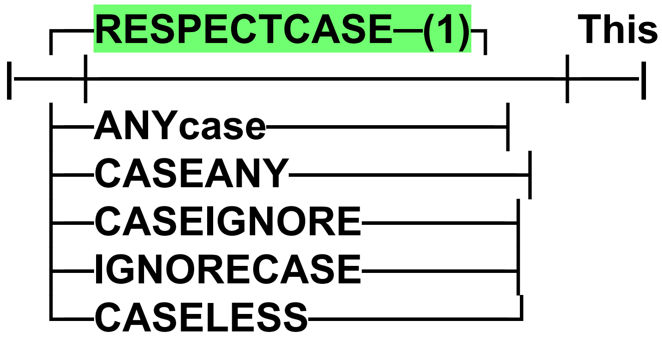
Added:

NetRexx /delimited_string/
now arbitrary NetRexx code for conversion and more

Prompts for defaults

Added to several stages, but not yet all, mostly for reading documentation

anycase:



This optional name for the default has been added

- **Reading Pipeline's "railroad" diagrams:**
  - **Mainline: Required**
  - **Below: Option**
  - **Above: Default option**

New NetRexx Stages

- Some new stages use NetRexx Rexx Power
- Some others harness Java/Unix style (but that is for another symposium – it is down the corridor – this one is Rexx)

Simple PARSE Stage

To break out parts of records

►►──PARSE──parse_template_Dstring──output_template_Dstring──►

- **Standard Rexx PARSE template, but variables are “_n”**
- **Standard Rexx clause for output**
- **Separated into Delimited strings, as from XEDIT**

Examples:

```
parse /. . _3 ./  /The third word is _3/

parse @_2 . =57 _1 +5@ !Part _1, has _2 units!
```

Full PARSE Stage

```
►►──PARSE──parse_template_Dstring──►

►──┬─────────────────────────────────────────────────────┬──►
   └───┬──NETREXX──┬──NetRexx_statement_Dstring──┘
       └──NR───────┘

►──┬──────────────────────────────────┬──►
   └output_template_Dstring──┘

►──┬─────────────────────────────────────────────┬──►◄
   └──FINALLY──NetRexx_statement_Dstring──┘
```

Note: For Java/Unix compatibility, there are GREP (REGEX) stages for similar actions. And there is the existing SPECs stage, which is based on a 1940s "wire programmed," punched card machine. All three stages have specific "strong points."

Simple SELECT Stage

►►──SELECT──T/F_NetRexx_Dstring──►◄

Note: "rec" is the current record, "prev" is the previous one

Examples:

select /parse rec 20 r +1; return r = 'B'/
This looks at a specific column, if it is the letter "B," the record is selected.

select /parse rec 2 r +1; parse prev 2 p +1; return r \= p/
This looks at a specific column, if it different from the preceding record, the record is selected.

MULTIple SELECT Stage

- Adding the option MULTIple allows up to a 10-way split
- In this case, the Dstring returns a digit from 0 to 9
- The object is directed to the numbered stage output

MULTI-way SELECT

Example: Translating a word-value into a digit for further processing

- **Here by the value of the second word**
- **("," is the line continuation character in pipelines)**

```
select multiple ,
      /parse rec . C .; ,
       r = 'blue red white'.wordpos(c); ,
       return r/
```

This looks at the second word and searches for it in a string of words; it reports the word position number, or 0 if not found.

MULTI-way SELECT Pipeline

```
pipe (m2)
 o: faninany |                This is for output, called later
   console ?
   literal green blue white |     Data input. It could be
   literal blue white green |     read from a file or array.
   literal brown red white |      It could be
   literal blue brown red |       Java/NetRexx objects
```

MULTI-way SELECT

```
  m: select multi ,
 /parse rec . c .; ,
 r = 'blue red white'.wordpos(c); ,
 return r/ |
```

Again,this looks at the 2nd word and converts it to a number.

MULTI-way SELECT

change //unknown color! / | o: ? (0 goes here)

m: | change //blue! / | o: ? (1 goes here)

m: | change //red! / | o: ? (2 goes here)

m: | change //white! / | o: ? (3 goes here)

(and all then goto o: for output)

MULTI-way SELECT

```
PS pipes> pipc m2
```

Compile the Pipeline to a Java class file

```
PS pipes> java m2
```

Run the Java class file.

Results:

```
unknown color! blue brown red
red! brown red white
white! blue white green
blue! green blue white
PS pipes>
```

Full SELECT Stage

```
              ┌─┬──TRUEfalse──┬─┐
              │ └──TF─────────┘ │
►►──SELECT────┼─────────────────┼────┬─T/F_NetRexx_Dstring───┬──►◄
              └─┬──MULTIple───┬─┘    └─Digit_NetRexx_Dstring─┘
                └──NOT────────┘
```

Custom Stages Are Needed Less

- CMS Pipelines has always allowed custom stages written in Rexx
- NetRexx Pipelines allows custom stages written in NetRexx
- These require some not so obvious coding for reading and writing records and for error trapping.
-
- Now the PARSE and SELECT stages, with a few lines of NetRexx code added, can eliminate the need for many custom stages.

Two Main Functions of Stages

- Select some records and not others
  - **ALL**
  - **BETWEEN**
  - **FIND**
  - **FROMLABEL**
  - **FRTARGET**
- Break some information out of records, maybe combine some
  - **CHANGE**
  - **CHANGEPARSE**

Break Text Records: PARSE Stage

- Easy to use Rexx notation to break records
- Two parts:
  - Standard Parse Template (but only _0 to _9 variable names)
  - Standard Rexx Clause (uses those variables)
- Separated as Xedit's Delimited Strings

```
parse / . _2 . 'foo' _7/  / 'Foo says:' _7 'in' _2/
```

PARSE Stage with NetRexx

- Can manipulate the variables in between
- Use a "NetRexx" ("NR") Delimited String

```
parse / . _2 . 'foo' _7/ ,
      nr / _3 = 3 * _2; _6 = _7.upper/ ,
      / 'Foo says:' _6 'in' _3/
```

PARSE Stage with Sumation

- A FINALLY Delimited String Adds A Record At The End
- The variable COUNTER[] persists, and is initialized to 0

```
parse / . _2 . 'foo' _7/ ,
         Nr /counter[1] = counter[1] + _2/ ,
         / 'Foo says:' _7 'in' _2/ ,
        Finally /'Total Foos:' counter[1]/
```

Pipes in NetRexx

- Pipes can be embedded in standard NetRexx
- Can read and write NetRexx variables and arrays
- Use the STEM stage for Arrays of objects
-
- NetRexx Pipelines actually works with full Java objects (mostly, and by default, text records of type Rexx)

Pipes in NetRexx

- Here is a full NetRexx program using an interior Pipeline.
- First, in NetRexx we define the Class, Method, and method properties

```
class testpipe
method testpipe(avar=Rexx)
   F = Rexx 'abase' -- Two NetRexx stem variables of objects
   T = Rexx 1       -- with default type Rexx, we needn't specify it
```

Pipes in NetRexx

```
F[0]=5          -- Fill the first stem variable
F[1]=222
F[2]=3333
F[3]=1111
F[4]=55
F[5]=444
```

Pipes in NetRexx

Now call the the pipe reading the stem variable

```
pipe (apipe stall 1000 )
  stem F |          -- read a NetRexx array to objects (records) |
    sort |                      -- sort column-wise  |
    prefix literal {avar} | -- show we can import variables |
    console |                   -- show them sorted |
    stem T
```

Pipes in NetRexx

```
-- Back in NetRexx, the second stem variable now has the sorted
-- objects (records), so show them
  loop i=1 to T[0]
    say 'T['i']='T[i]
  end
```

Pipes in NetRexx

```
-- And drive the whole thing, grabbing a system argument for all the
-- way into the pipe.
method main(a=String[]) static
  testpipe(Rexx(a))
-- Remember from above we have
--    method testpipe(avar=Rexx)
-- And the stage
--    prefix literal {avar}
```

Compile the NetRexx using the Pipe Compiler

```
PS pipes> pipc testpipe.nrx
pipe (testpipe_apipe stall 1000 ) stem F | sort | prefix literal
arg(string {'avar'}) | console |  stem T
```

And Run the resulting Java .class file

PSpipes> **java testpipe jeff** with a parameter

```
jeff
1111
222
3333
444
55
```

Results back in NetRexx

```
T[1]=jeff
T[2]=1111
T[3]=222
T[4]=3333
T[5]=444
T[6]=55
```

Nice, but the Sort is by "Card Columns"

- The SORT stage follows the CMS convention of sorting by columns by default.
- For "Rexxification,"
- NetRexx Pipelines adds the option of TOLERANT to use Rexx sorting, so numbers are in numerical order (and leading spaces are ignored in strings).
- So adding that option gives this pipe:

Using Rexx, Tolerant, Sorting

```
pipe (apipe stall 1000 )
    stem F | -- read a NetRexx array to objects (records) |
    sort TOLERANT | -- sort Rexx-wise  |
    prefix literal arg(string {'avar'}) |
    console |     -- show them sorted |
    stem T
```

Tolerant SORT Compiled and Run

```
PS pipes> pipc testpipe.nrx
pipe (testpipe_apipe stall 1000 ) stem F | sort tolerant | prefix literal arg(string {'avar'}) | console | stem T
PS pipes> java testpipe jeff
```

Tolerant Console Results

```
jeff
55
222
444
1111
3333
```

Now sorted in true numerical order. (The SAY statement could have formatted them to be right justified.)

Tolerant Array Results and Comparison

[Tolerant]	[Strict (default)]
T[1]=jeff	**T[1]=jeff**
T[2]=55	**T[2]=1111**
T[3]=222	**T[3]=222**
T[4]=444	**T[4]=3333**
T[5]=1111	**T[5]=444**
T[6]=3333	**T[6]=55**

Rexxifying the SORT Stage

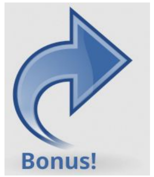
```
►►──SORT──┬─────────────────────────────────────────┬───────┬──────────────┬────►
          │     ┌─REXX──────┐     ┌─10000──┐         │       └─inputRange───┘
          └─(─┼───────────┼─────┼────────┼─)─┘
               └─class─(2)┘   └─size────┘

     ┌─Ascending─(1)──┐                                   ┌─STRICT─(4)───┐
►──┼────────────────┼──┬────────────────┬──┼──────────────┼──►◄
     └─Descending─(1)┘  └─SINGLEOK─(3)┘  └─TOLERANT─(5)┘
```

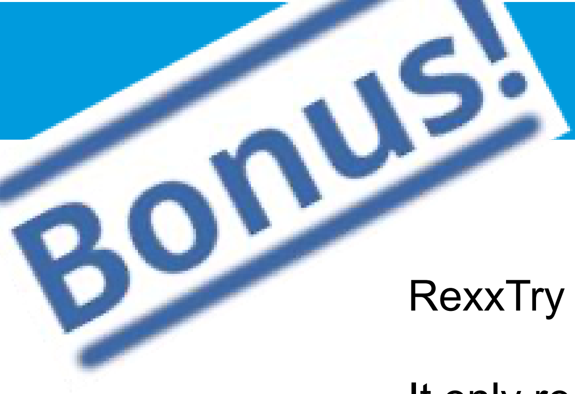
Three Considerations for Rexxification

1) Existing Stages
2) New Stages
3) Using a pipe in NetRexx

Goodbye!

- Rexxifying Pipelines
  - Makes it easier to read and maintain
  - Makes it easier to write
- You should consider including Pipelines in your set of Rexx tools
-
- The End
- Jeff@Jeff-H.com





RexxTry

RexxTry is one of the oldest Rexx programs.

It only recently was able to be implemented in NetRexx

Like other versions, it allows immediate single line execution of Rexx code

NetRexx adds multi-line execution by adding prefixed line numbers

Can be traced and single-stepped, à la "TRACE ?" (Which is missing from NetRexx.)

RexxTry in NetRexx

```
> nrc -exec rexxtry
===== Exec: rexxtry =====
  NetRexx 5.00 11 Dec 2024
  rexxtry lets you interactively try NetRexx statements.
    Each string is executed when you hit Enter.
    Enter 'tell' for a description of the features.
  Go on - try a few...          Enter 'exit' to end.
Ready
>-
```