# From BREXX to BREXX/370

## The Journey

An evolution nobody asked for – but we did it anyway

Peter Jacob

Decommisioned, not Deprecated

# “REXX reads like English, runs like C, and debugs like a dream”

*Yes, I know — I’m preaching to the choir.*

# The House of BREXX

**Vasilis Vlachoudis, PhD**

Physicist at CERN

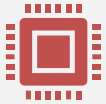
Inventor of BREXX

**Mike Großmann**
by day
Beta Systems Software AG

by night:
co-migrator, chaos-tamer,
and the reason BREXX/370 still talks to us.

Me — retired, but only from employment,
not from strange ideas

# BREXX Revisiting the Origins



In 2001, **Vasilis Vlachoudis** presented **BREXX** at this conference, introducing a compact, efficient REXX interpreter for various **non-mainframe platforms** —a model of clarity and portability.

His architecture provided a robust starting point.

**While BREXX/370** targets the **IBM mainframe environment**, it still builds directly on the structure and principles he established.

Before we dive into the new features, let’s take a quick look at a few of his original foils —
the roots of what has now evolved into **BREXX/370**.

# Vasilis Vlachoudis

# Author of BREXX

Studied Physics at **Aristotle University of Thessaloniki** (1987–1991), where he first encountered REXX on a VM/370 system — an experience that sparked a lifelong interest.

After struggling to find a functional REXX interpreter for **MS-DOS**, he began developing his own from scratch.

During his PhD at **CERN** (1996), he rewrote and optimized the interpreter, resulting in **BREXX**, originally designed for Monte Carlo data processing.

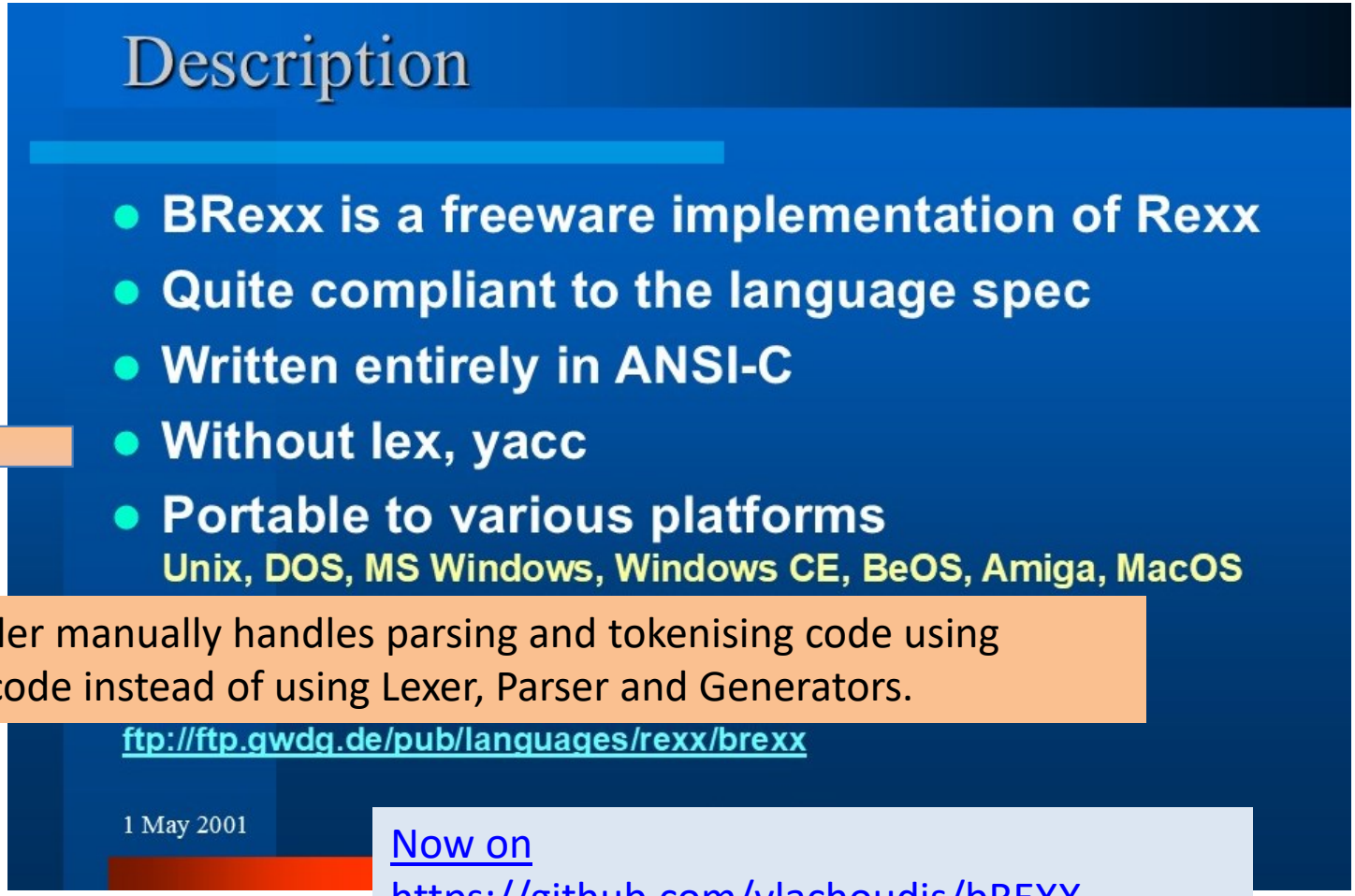
Today, he is a **Senior Physicist at CERN**, leading the **FLUKA** and **FLAIR** projects — while his contribution to REXX lives on through the BREXX project.

The core REXX language and processing remained unchanged

Other functionality and enhancements were provided as MVS-owned add-ons.

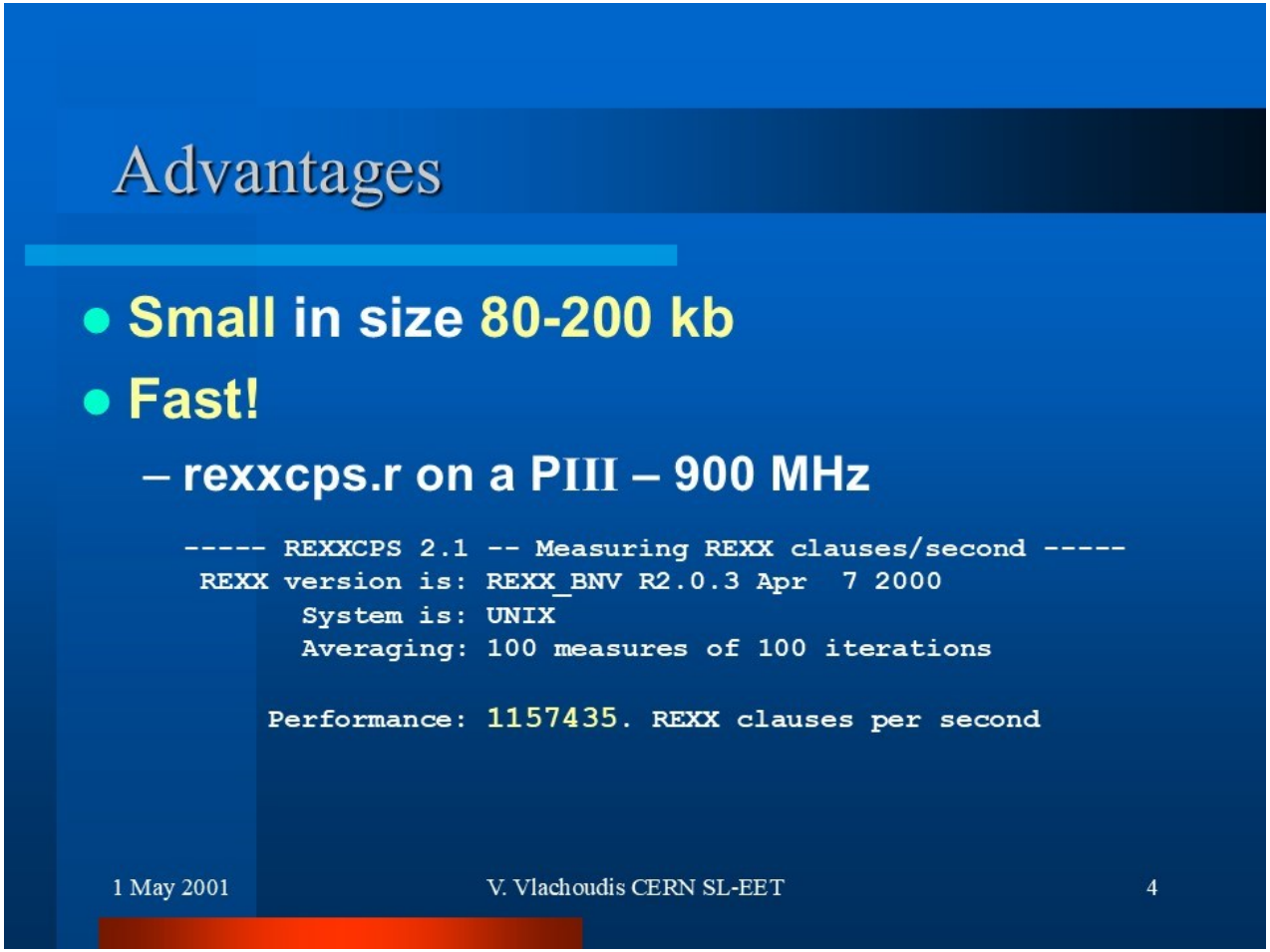
*His original architecture continues to inspire — and serve as the foundation for BREXX/370.*


From BREXX to BREXX/370: The Journey




The compiler manually handles parsing and tokenising code using custom C code instead of using Lexer, Parser and Generators.

Now on https://github.com/vlachoudis/bREXX




From BREXX to BREXX/370: The Journey

# History

- 1991 First attempt
- 1992 V1.0 Buggy
- 1994 V1.3 Quite stable, slow!
- 1998 Started working in
- 1999 V2.0 New code, fas
- 2001 V2.0.3 Current Ver



# What changed from V1 to V2

- V1 interpreting on the fly
  - Compact code
  - Slow
- V2 compiles the Rexx code and then it interprets the compiled code.
  - Optimized (ie. use of caching for variable access, …)
  - more than 10 times faster
  - Needs slightly more memory for execution



From BREXX to BREXX/370: The Journey

# Differences with ANSI Rexx [2/2]

- **Not implemented in BRexx:**
  - *CALL ON <…>*
  - *NUMERIC DIGITS* [up to 15 digits]
    - All floating point operations are using doubles
  - *NUMERIC [FORM | FUZZ]*
  - *TRACE RESULTS* = INTERMEDIATES



# Differences with ANSI Rexx [1/2]

- **Existing in BRexx and not in ANSI:**
  - BRexx I/O specific
    - *open, close, read, write, eof, flush, seek*
      [ All ANSI I/O functions are supported also ]
  - Math functions
    - *sin, cos, exp, pow, …*
  - VM/CMS Stack
    - *"dir (stack"*
  - *LOAD()* – importing external libraries
  - Platform depended functions



From BREXX to BREXX/370: The Journey

# Variable Structure

- The variables are stored with a "hashlist" pointing to optimized binary trees.
- There is a caching mechanism for accessing the variables.
- Every procedure with private variable scope, creates a new variable pool.
- Pools are indexed:
  - 0 The main code
  - 1 First procedure
  - 2 Second
  - ...
- VALUE(name[,[newvalue][,p



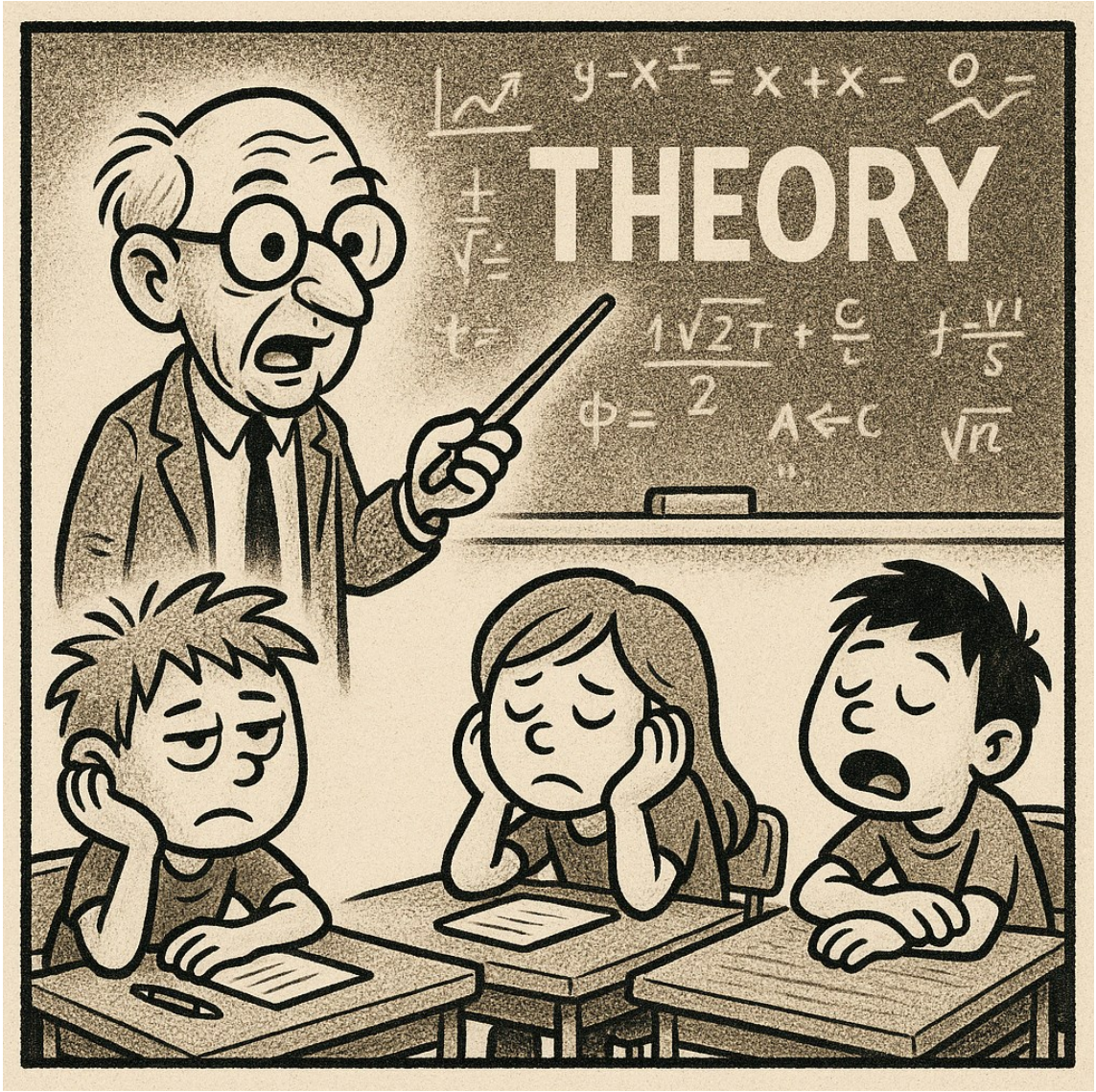
# Variable Storage

- Each variable in BRexx is stored as a length prefix string, with 3 different types, integer, real, string depending on the last operation on the variable.
  - a = "2.0" String
  - a = "2" Integer [32 bit]
  - a = a + 1 Integer [32 bit]
  - a = a + 0.1 Real [64 bit]
  - substr(a,2,1) String
- All the conversions are transparent to the user



From BREXX to BREXX/370: The Journey

# Why Mainframes Still Matter



# Why Mainframes Still Matter

| Aspect | Mainframe (e.g., IBM z/OS) | PC / Distributed Systems |
|---|---|---|
| **Purpose** | High-volume, mission-critical processing | General use / horizontal scaling |
| **Uptime** | Designed for near-100% availability (99.999%) | Acceptable downtime; built-in failover needed |
| **Users** | Thousands of concurrent users on one machine | Usually 1 user per PC; distributed uses many |
| **Performance** | Massive I/O throughput, batch & transaction-heavy | Great at parallel tasks, web, microservices |
| **Security** | Centralized, highly secure environments | Varies; more exposed surface across nodes |
| **Longevity** | Runs decades-old apps reliably | Shorter lifecycle; frequent upgrades |

# Performance: It’s Not Apples to Apples

**It’s not about clock speed — it’s about what the system is built to do**

### 🧠 z16 Mainframe CPU

- Built for throughput & reliability
- Delegates to specialised processors
- Optimised for 24/7 enterprise-scale workloads
- Secure, consistent, highly scalable

### ⚙️ Commodity CPU

- Built for speed & versatility
- Handles most tasks internally
- Great for short, bursty workloads
- Distributed, flexible, modular

🏁 **A z16 doesn't try to win the GHz race — it wins the enterprise marathon.**

# Performance:
# It’s Not Apples to Apples

- **IBM z16 Processor (Telum chip):**

- 8 cores per chip
- Up to 5.2 GHz

- **Commodity CPUs (e.g. Intel Core i9-13900K):**

- 24 cores (8 P + 16 E)
- Up to 5.8 GHz

- **Mainframes CPUs are designed for**
  - throughput
  - system-level reliability,
  - massive parallelism across enterprise-scale workloads.
- **z16 Advantages Beyond Raw Speed:**
  - Simultaneous Multi-Threading (SMT) optimised for I/O-heavy workloads
  - On-chip AI inference with sub-millisecond latency
  - Cache and memory architecture tuned for sustained high-load performance
  - Vertical scaling up to 200+ cores across drawers, all managed as a single system

# Mainframes: The Central Point of Control

## 🧠 Mainframe CPU – The Conductor

Delegates tasks to assist processors

- 💾 SAPs: I/O operations
- 🐧 IFLs: Linux workloads
- 📦 zIIPs/zAAPs: DB2, Java, XML
- 🔐 Crypto cards: encryption

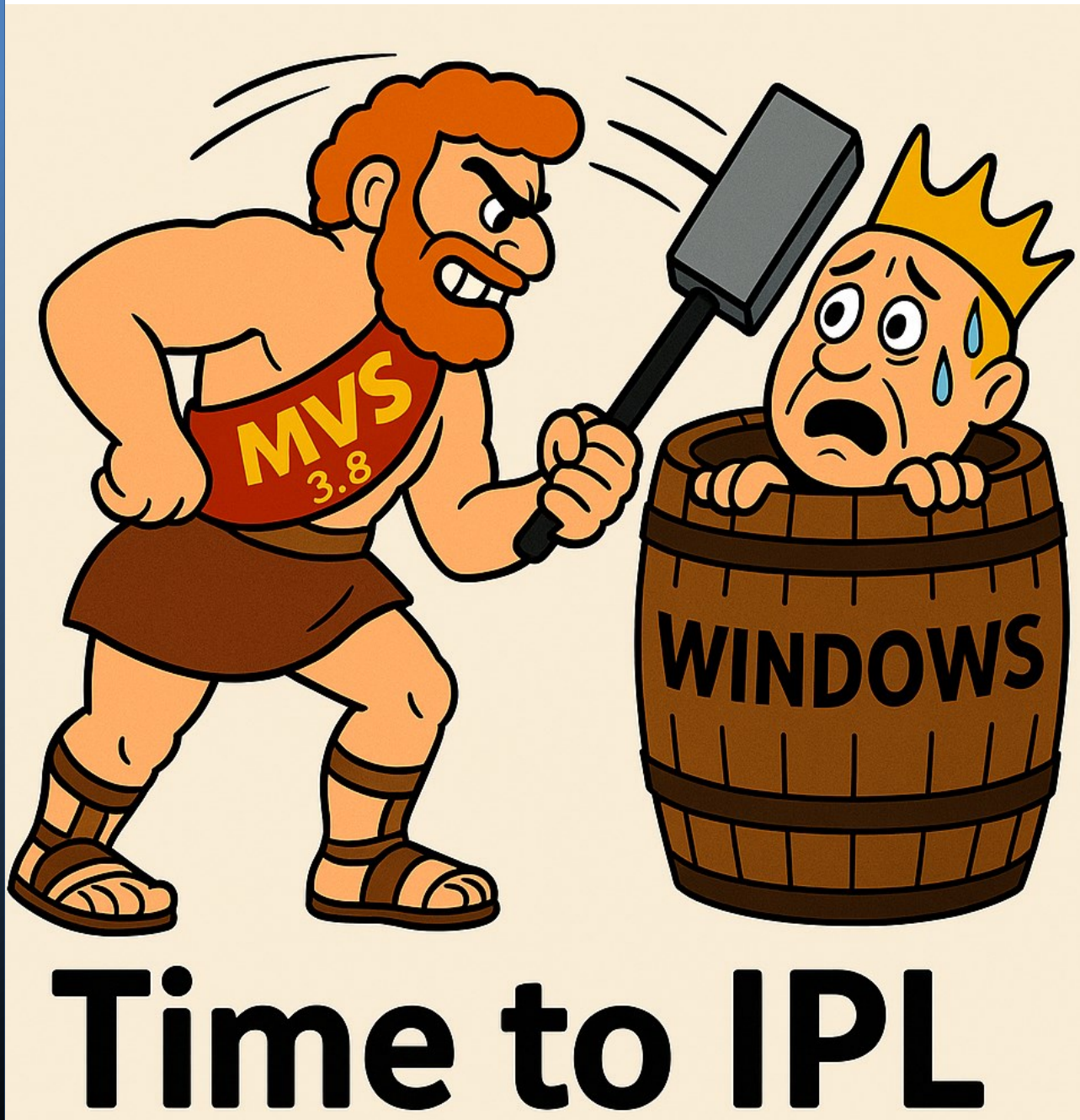
Enables high throughput & reliability

## 🎻 Commodity CPU – The Soloist

- Handles most tasks internally
- Less delegation,
- general-purpose logic
- Optimized for parallel user workloads
- Requires distributed system for scaling
- More moving parts, more management

The Great ⚙ IPL Awakening:

**All theory, now let it roar. Loud.**



# From BREXX to BREXX/370 The Journey

**Compiler: JCC (MVS 3.8 C Compiler)**

- Used for **BREXX migration**
- Built specifically for **MVS 3.8 environments**
- **Not a modern development toolchain**
- **No longer actively maintained**

**C Language Support**

- Supports only **C89 (ISO/IEC 9899:1990)**
- Lacks support for modern C standards (C99/C11)
- Severely **restricts maintainability and modernization**
- Outdated features and tooling introduce **development overhead**

**Library Limitations**

- **Memory management worked — until it didn't**
- Standard C Library: Incomplete
- Missing or inconsistent libc functions
- No support for POSIX or third-party libraries
- Forces custom, time-consuming implementations

It seemed like a good idea. At the time. In the fog.

# From BREXX to BREXX/370: the Journey

**First Release 1. April 2019**

**Several Releases since then, the current release V2R5M3**

**Delivered Documentation:**

- **Installation document**
- **BREXX/370 User's Guide**
- **Formatted Screens User's Guide**
- **VSAM User's Guide**
- **BREXX Arrays**

**Releases available at:** https://github.com/mvslovers/brexx370/releases

**Original BREXX available at:** https://github.com/vlachoudis/brexx

# Performance MVS % Windows

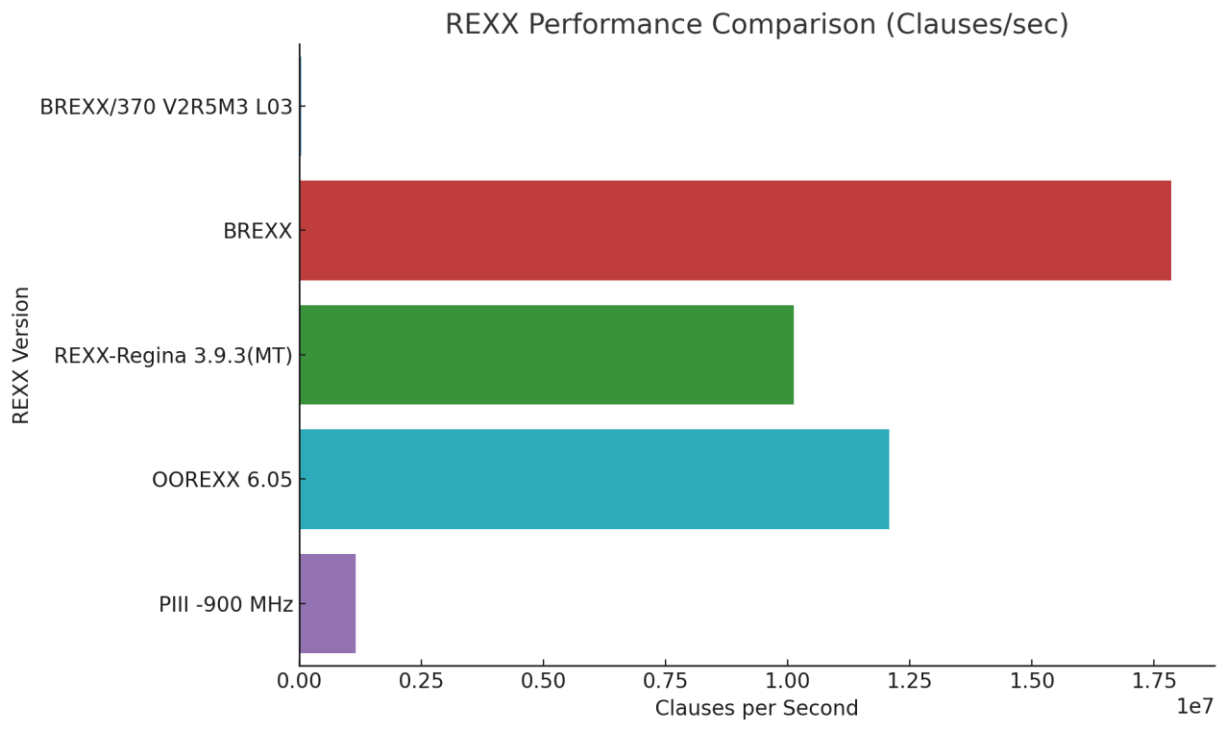
```
----- REXXCPS 2.1 -- Measuring REXX clauses/second -----
 REXX version is: BREXX/370 V2R5M3 L03 (Sep 24 2024)
       System is: MVS
       Averaging: 100 measures of 100 iterations

     Performance: 43077 REXX clauses per second

  BREXX Statistics
      Instructions 8041038
      Elapsed Time     232.267015 secs
      Instructions   34619.801696/secs
 ***
```

```
----- REXXCPS 2.1 -- Measuring REXX clauses/second -
 REXX version is: BREXX/370 V2R5M3 (Jan 29 2025)
       System is: UNIX
       Averaging: 100 measures of 100 iterations

     Performance: 17847138 REXX clauses per second

  BREXX Statistics
     Instructions 8041038
     Elapsed Time      1.016517 secs
     Instructions 7910514.510575/secs
```

On Windows and Linux, BREXX/370 delivers performance up to 100x faster than its MVS counterpart.

# REXX Performance Comparison

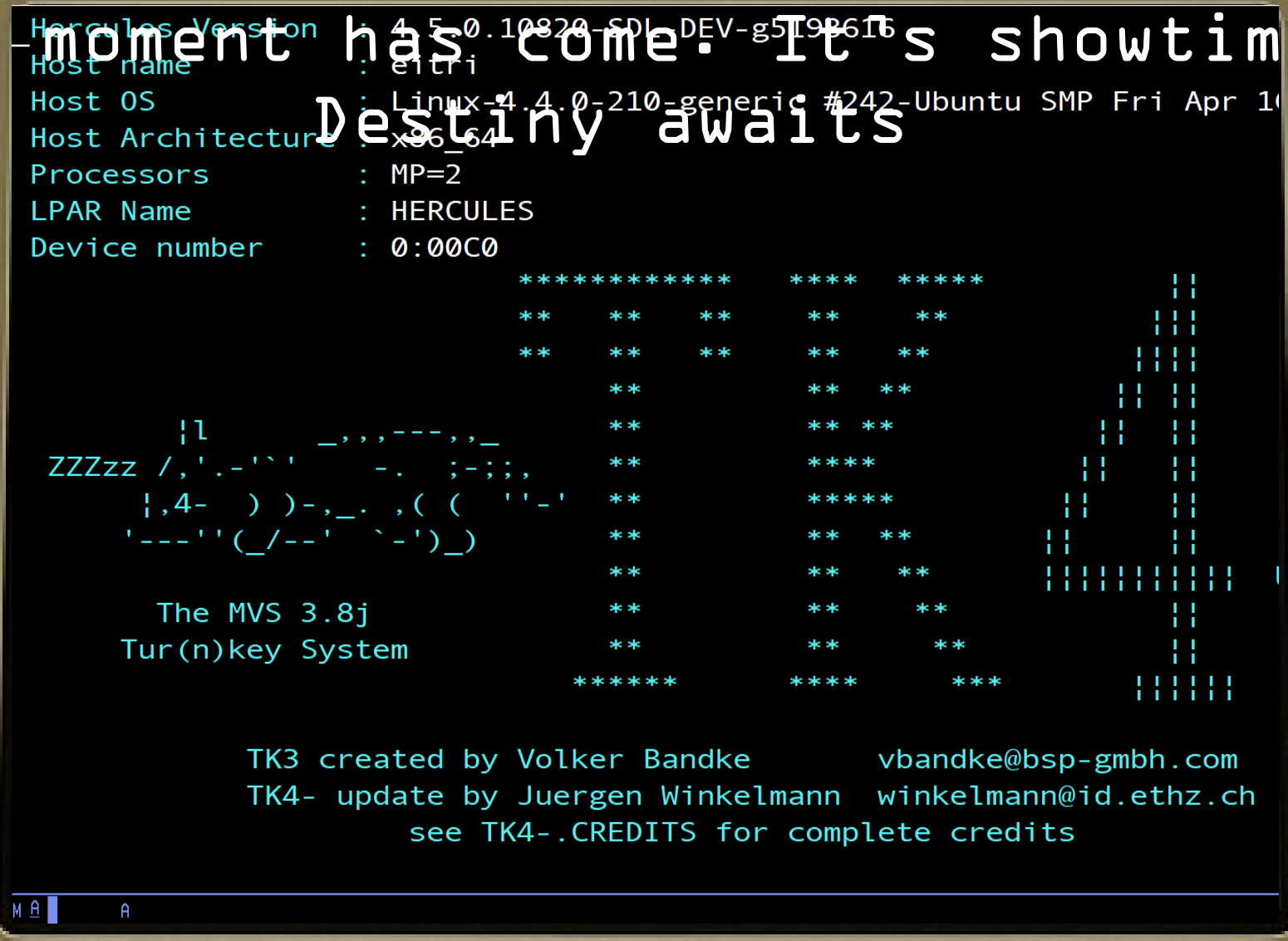
| REXX Version | System | Date | Clauses/sec | Elapsed Time (s) | Instructions/sec |
|---|---|---|---|---|---|
| BREXX/370 V2R5M3 L03 | MVS | Sep 24, 2024 | 43,077 | 232.267 | 34,619.80 |
| **BREXX** | **UNIX** | **Apr 16, 2025** | **17,847,138** | - | - |
| REXX-Regina 3.9.3(MT) | WIN64 | Oct 5, 2019 | 10,122,503 | - | - |
| OOREXX 6.05 | WindowsNT | Jun 25 2024 | 12,077,295 | | |
| **PIII -900 MHz** | **Unix** | **Apr 7, 2000** | **1,157,435** | | |




The moment has come. It's showtime.
Destiny awaits.
Hercules Version : 4.5.0.10820-SDL-DEV-g5198616
Host name : eitri
Host OS : Linux-4.4.0-210-generic #242-Ubuntu SMP Fri Apr 16 09:57:56
Host Architecture : x86_64
Processors : MP=2
LPAR Name : HERCULES
Device number : 0:00C0
ZZZzz
The MVS 3.8j
Tur(n)key System
Update 08
TK3 created by Volker Bandke vbandke@bsp-gmbh.com
TK4- update by Juergen Winkelmann winkelmann@id.ethz.ch
see TK4-.CREDITS for complete credits

The moment has come. It's showtime.

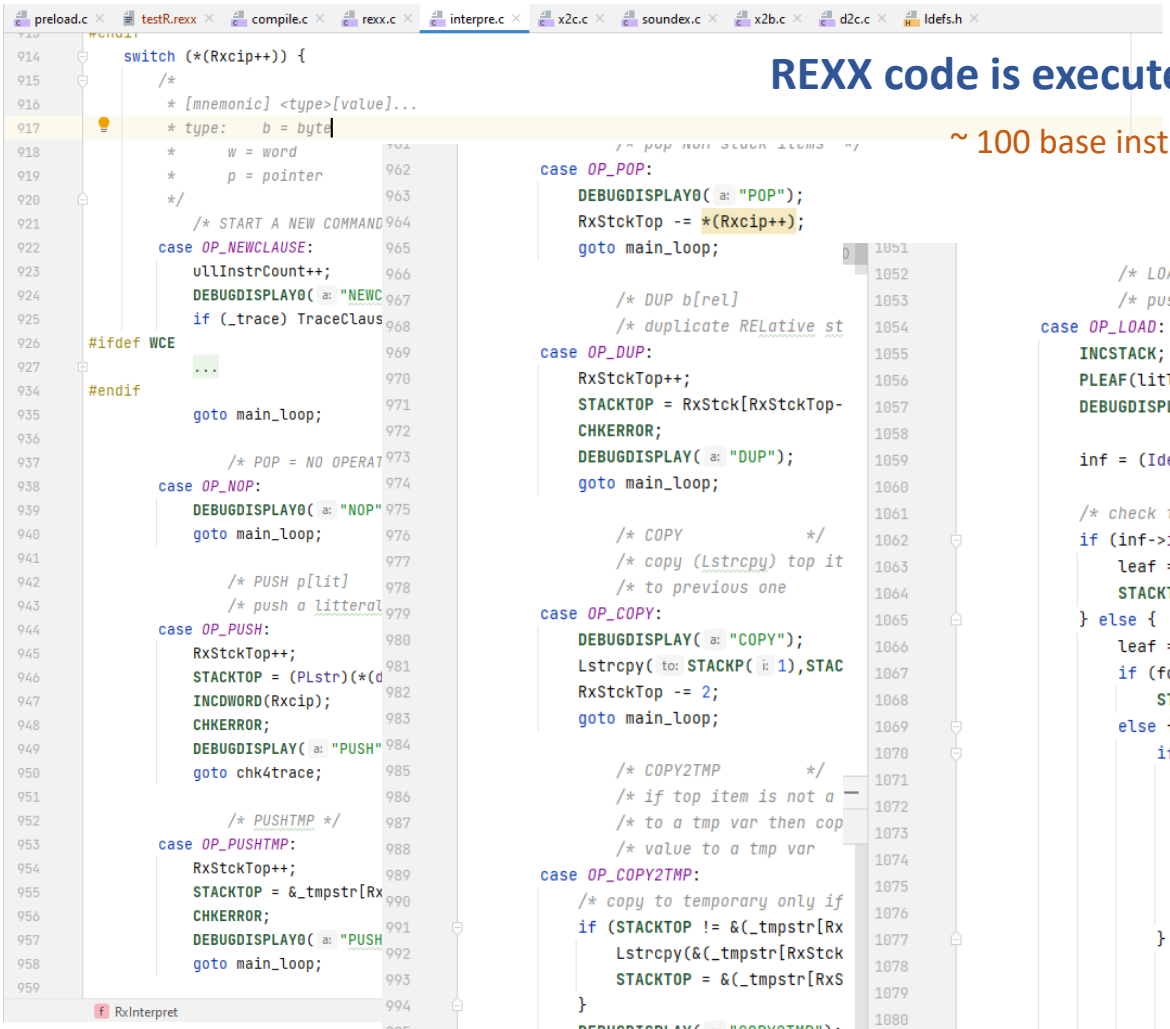
Destiny awaits




The cursor blinks. The system awakens!

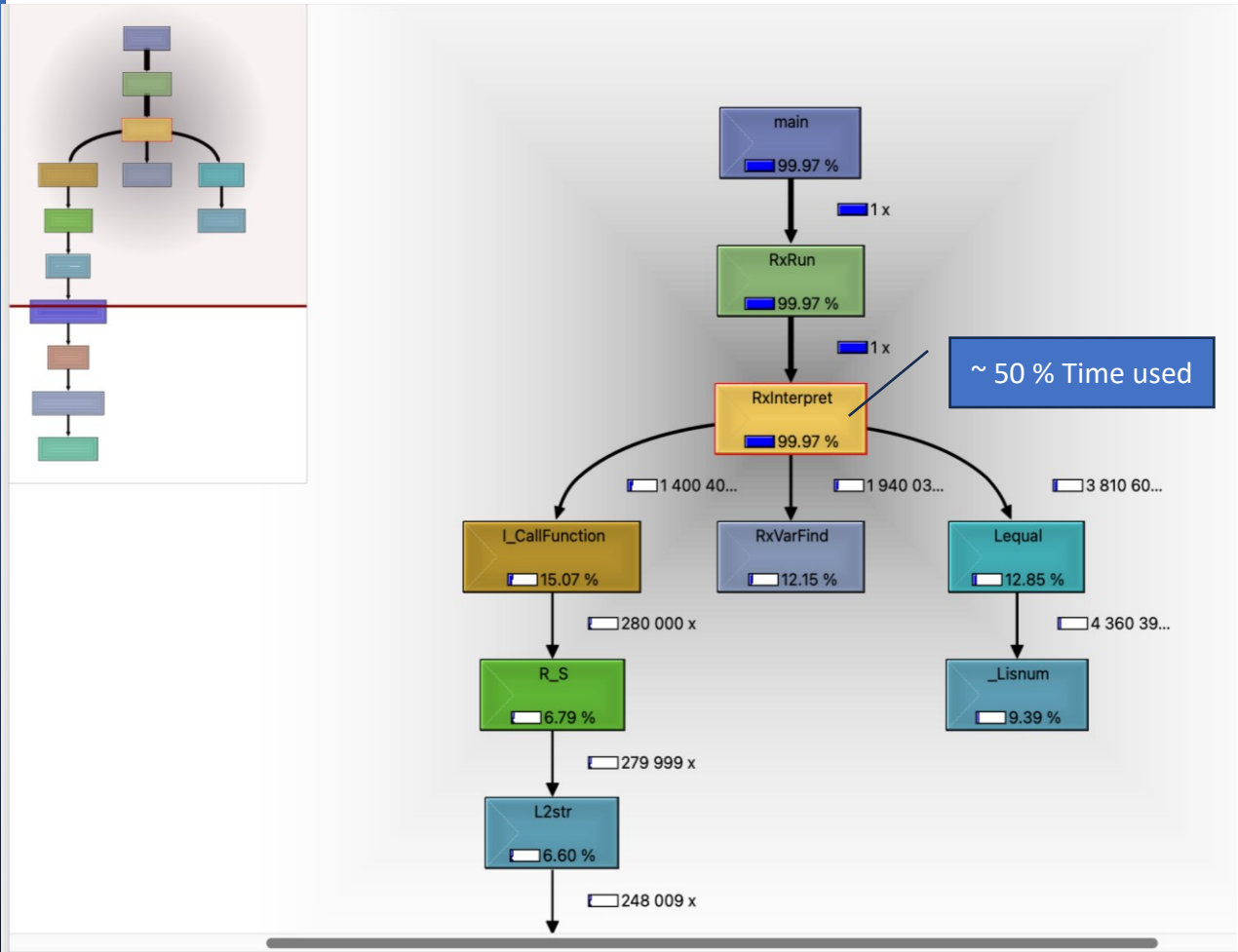
# REXX code is executed via a streamlined token stream

~ 100 base instructions



# Where time is consumed

(CALGRIND)



# Now Off-Road

```
Entering the wilderness...
```

# BREXX‘s Distinct Characteristics

Good performance

Pre-compiled (tokenising at run time)

Split into elementary tokens
Tokens placed into a **Token Stream** (array)
Variables and STEMs resolved via binary tree lookup
Compiled into stack operations using Reverse Polish Notation (RPN)

Execution runs the **Token Stream**

Benefits / Disadvantages

The Token Stream is live – and it may grow as needed
Labels must be unique over the Token Stream
Call of procedures (labels) work within the stack
This allows call-back procedures
Variables unique and accessible/maintainable in all Procedures (without PROCEDURE statement)

Pre-compiled % Compiled

# BREXX approach

- BREXX takes a direct, integrated approach to compilation
- No separate lexer, tokeniser, or parser stages
- No abstract syntax tree (AST)
- Tokenisation directly drives bytecode generation
- Simplified structure supports minimal tooling and fast execution

Pre-compiled % Compiled

# CREXX
## approach

- Lexer (Lexical Analysis)

- Tokeniser (Tokens)

- Parser (Syntactic Analysis)

- Grammar (REXX)

- Abstract Syntax Tree (AST)

- Code Generator

# CREXX Compilation Process

**Lexer (Lexical Analysis)**:

- Converts raw source code into lexemes (e.g., count = 10; → [count], [=],, [;]).
- Uses rules like regular expressions to identify lexemes.

**tokeniser (Tokenization)**:

- Converts lexemes into structured tokens (e.g., IDENTIFIER(count), EQUALS(=), NUMBER(10), SEMICOLON(;)).

**Parser (Syntactic Analysis)**:

- Builds syntactic structure from tokens using grammar rules.
- Example: assignment_statement → identifier '=' expression ';'.

**Abstract Syntax Tree (AST)**:

- Simplified tree representation of syntactic structure.
- Example: Assignment ├── Identifier: count └── Value: 10.

**Code Generator**:

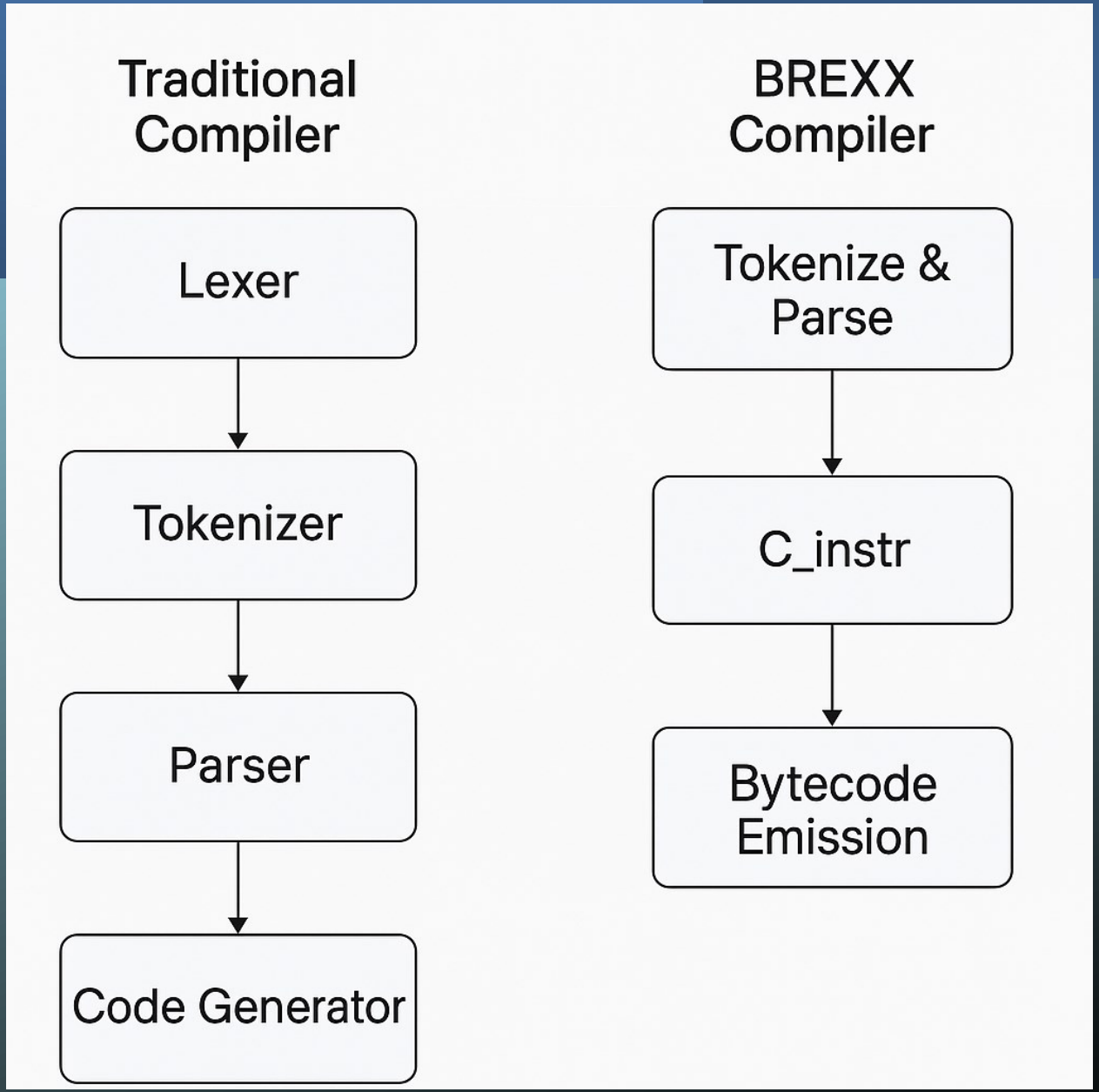
- Converts AST into CREXX assembly code.
- Example: count = 10 → MOV [memory_address], 10.

**Complete Workflow**:

- Source Code → Lexer → tokeniser → Parser → AST → Code Generator → Target Code.

# BREXX/CREXX

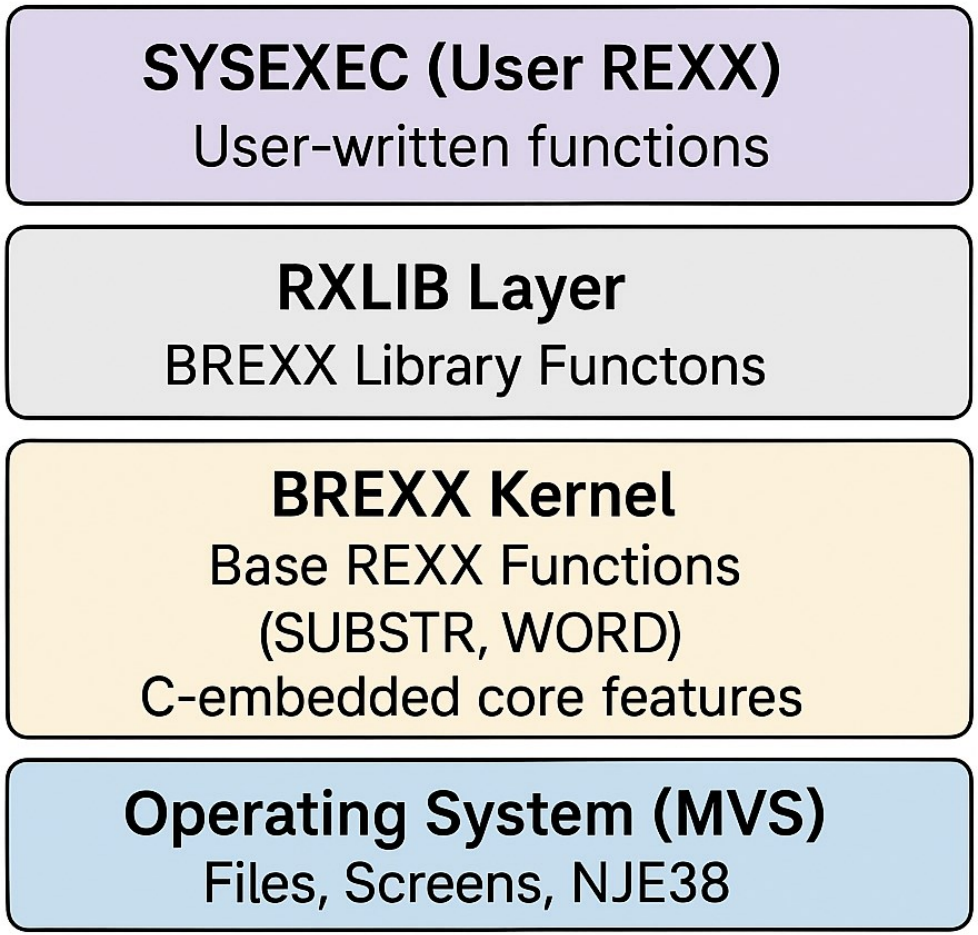
## Diverging Designs
## Shared Goals



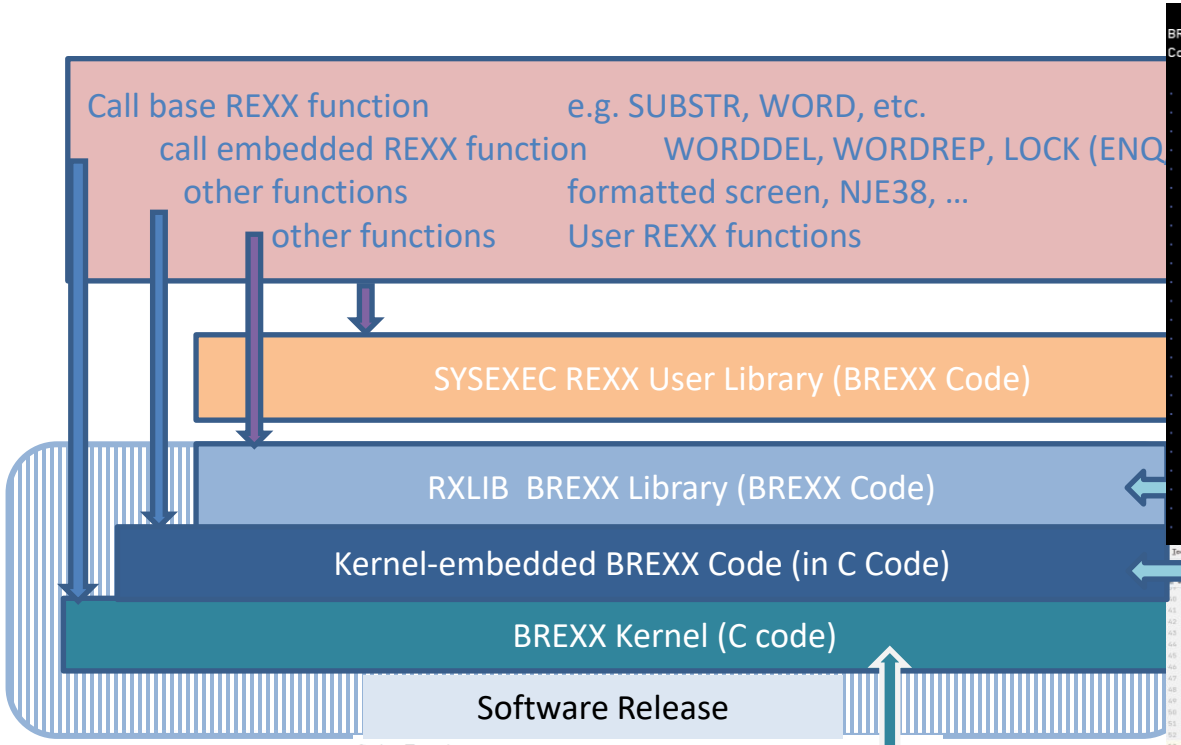
And now back from something completely different

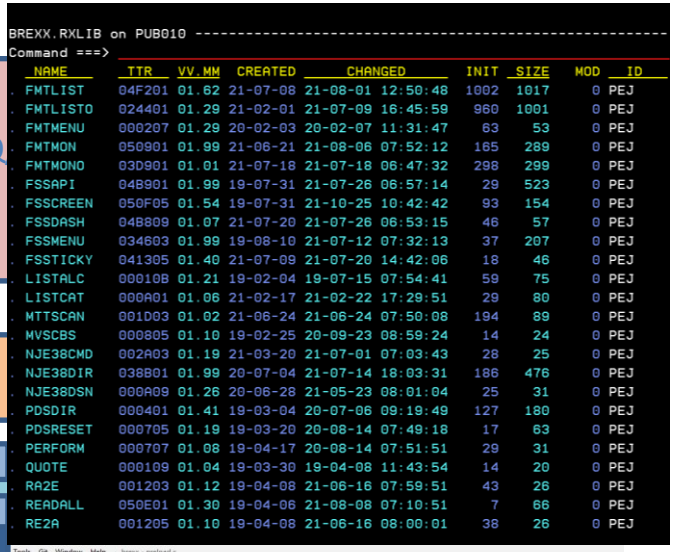
To
BREXX/370 Layer Concept

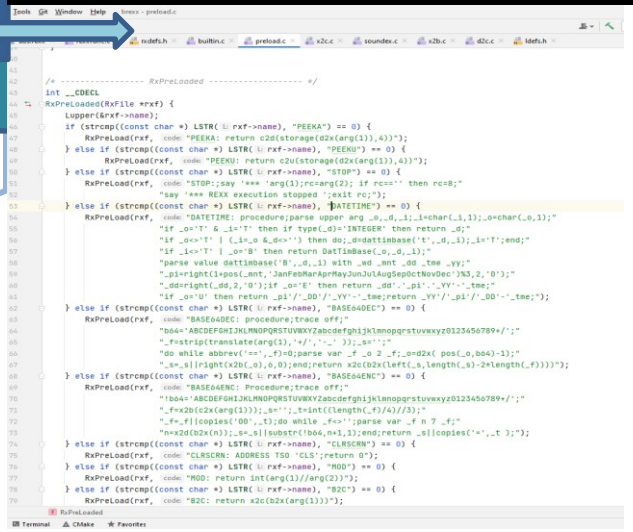
# BREXX Layer Concept

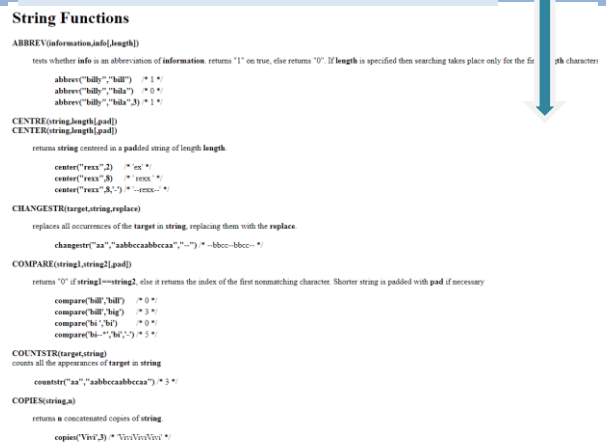


# BREXX Layer Concept



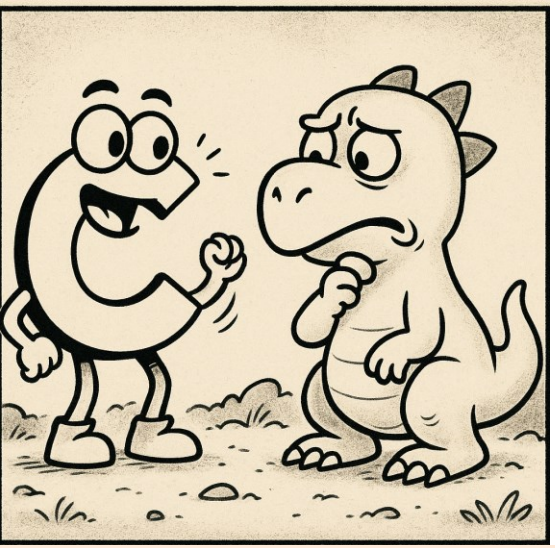





Transparent integration of REXX-written core utilities

# BREXX/370

## Embedded Kernel Functions



C now speaks REXX. Not fluently—but enough to get into trouble.

# Embedded REXX Functions

BREXX supports embedded kernel functions written in REXX code strings, which are transparently loaded and behave like native built-in functions.

- Defined in C via `RxPreLoad()` as string-based REXX procedures
- Automatically injected at runtime when referenced
- Appear native to the interpreter and user
- Allows definition of new procedures on the fly
- No visible distinction from compiled commands

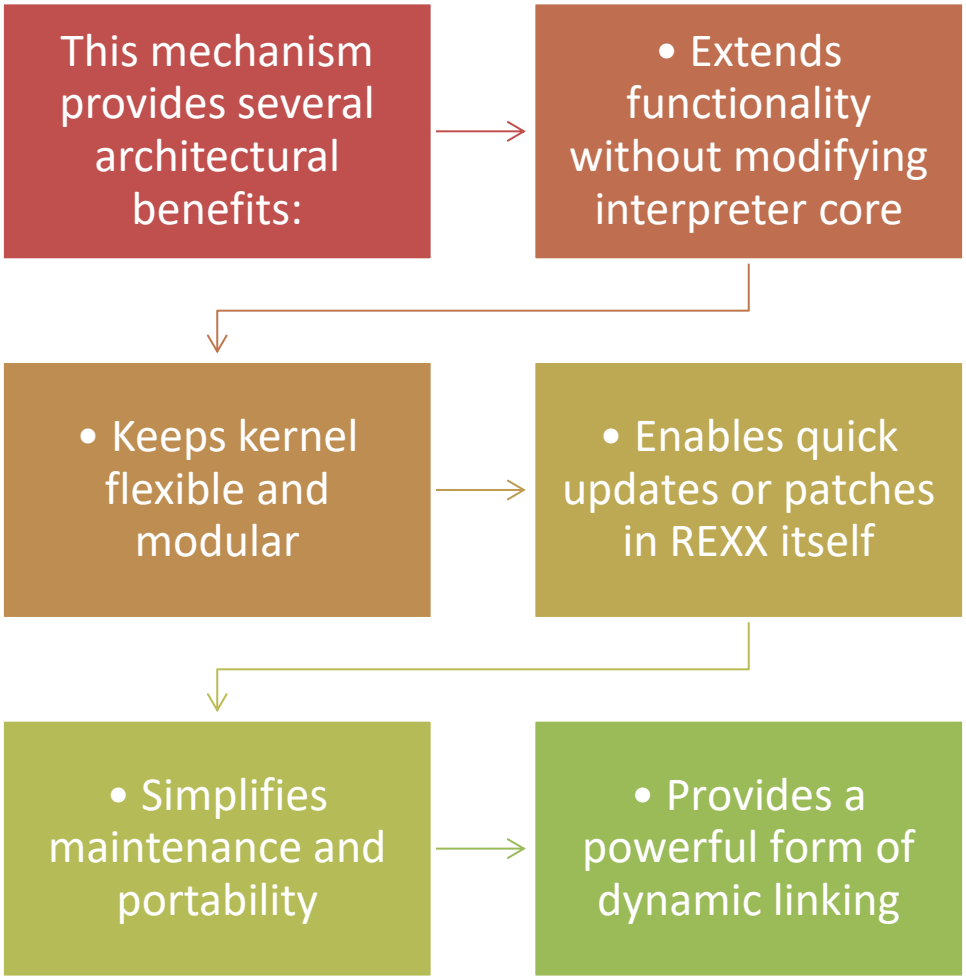
# Examples of Embedded Kernel Functions

These REXX functions are defined in `preload.c` and transparently loaded:

- PEEKA, PEEKU, PEEKN – memory inspection
- BASE64ENC, BASE64DEC – base64 utilities
- STEMCOPY, STEM2LL, LL2STEM – stem/list conversions
- DATETIME, EPOCH2DATE – date/time formatting
- JOBINFO, SYSVAR – system environment access
- MOD, ROOT, CLRSCRN – general-purpose utilities
- etc. …

# Embedded Kernel Functions

# Design Advantages



# Example **LISTVOLS** volumes of a **MVS System**

Deep inside the BREXX Reactor Core, this REXX module continues to deliver time-tested power.":

```
} else if (strcmp((const char *) LSTR(rxf->name), "LISTVOLS") == 0) {
   RxPreLoad(rxf,"LISTVOLS: Procedure  expose volumes.; trace off; parse upper arg option; call privilege('on'); call outtrap('dev.');"
            "ADDRESS COMMAND 'CP DEVLIST'; call outtrap('off'); call privilege('off'); bi=0; do volj=1 to dev.0;"
            "parse upper value word(dev.volj,4) with dasd'/'vol'.'dev; if dasd<>'DASD' then iterate; bi=bi+1;"
            "unit=word(dev.volj,3); buffer.bi=vol'  'unit'  'dev; end; bi=bi+1; buffer.0=bi; buffer.bi=bi-1' Volumes found';"
            "if abbrev('FMTLIST',option,3)>0 then call fmtlist ,,'Volume  Unit Device'; else if abbrev('LIST',option,3)>0 then do;"
            "say 'Volume  Unit Device'; do i=1 to buffer.0; say buffer.i; end; end; else do; buffer.0=buffer.0-1;"
            "do i=0 to buffer.0; volumes.i=buffer.i; end; end; return;");
```


The moment has come. It's showtime.
Destiny awaits.
Hercules Version : 4.5.0.10820-SDL-DEV-g5198616
Host name : eitri
Host OS : Linux-4.4.0-210-generic #242-Ubuntu SMP Fri Apr 16 09:57:56
Host Architecture : x86_64
Processors : MP=2
LPAR Name : HERCULES
Device number : 0:00C0
ZZZzz
The MVS 3.8j
Tur(n)key System
Update 08
TK3 created by Volker Bandke vbandke@bsp-gmbh.com
TK4- update by Juergen Winkelmann winkelmann@id.ethz.ch
see TK4-.CREDITS for complete credits

# BREXX/370

## Added Features part I

| | |
|---|---|
| ADDRESS FSS | Formatted Screen Services<br>Menus, Screens, Lists |
| **VSAM KSDS** | support read and write VSAM records |
| **ADDRESS LINK/LINKMVS/LINKPGM/ATTACH** | Call external programs |
| **ADDRESS MVS** | Interface to certain REXX environments as VSAM and EXECIO |
| **ADDRESS TSO** | Interface to the TSO commands, e.g. LISTCAT, ALLOC, FREE, etc. |
| **ADDRESS COMMAND** | Interface to the Host system of your MVS system. Typically Hercules or VM370 |

If it blinks, logs, or sighs—REXX can interface it.

# BREXX/370

## Added Features part II

**TCP/IP Interface** Building TCP servers as well as TCP clients

**Interface to NJE38** communicating with other MVS and CMS sites: messages and datasets

**Operator Interface** sending operator commands

**OUTTRAP** hijack write to terminal (say …)

**Receiving Master Trace Table events**

**Dynamic Dataset Service**
CREATE/ALLOCATE/FREE/OPEN (new) Datasets

**Create and execute REXX scripts on the fly**

**Global and Profile Variables** read/write variables across procedures

**New REXX Functions** many, many


The moment has come. It's showtime.
Destiny awaits.
Hercules Version : 4.5.0.10820-SDL-DEV-g5198616
Host name : eitri
Host OS : Linux-4.4.0-210-generic #242-Ubuntu SMP Fri Apr 16 09:57:56
Host Architecture : x86_64
Processors : MP=2
LPAR Name : HERCULES
Device number : 0:00C0
ZZZzz
The MVS 3.8j
Tur(n)key System
Update 08
TK3 created by Volker Bandke vbandke@bsp-gmbh.com
TK4- update by Juergen Winkelmann winkelmann@id.ethz.ch
see TK4-.CREDITS for complete credits

# MVS 3.8

# special functions

**RXMVS adds a large set of specialized functions** for BREXX that interface directly with **IBM MVS** and **TSO (Time Sharing Option)** environments

These functions are **not part of standard REXX**, but are tightly bound to:

MVS-style datasets and file structures (PDS, PS)

MVS system information blocks (CVT, PSA, SMCA)

TSO-specific features (console access, SVC calls, user IDs)

System Privilege Management (via Privilege, SYSVAR, etc.)

JES/NJE environment integration (e.g., SYSNJVER, RxNjeGetNetId)

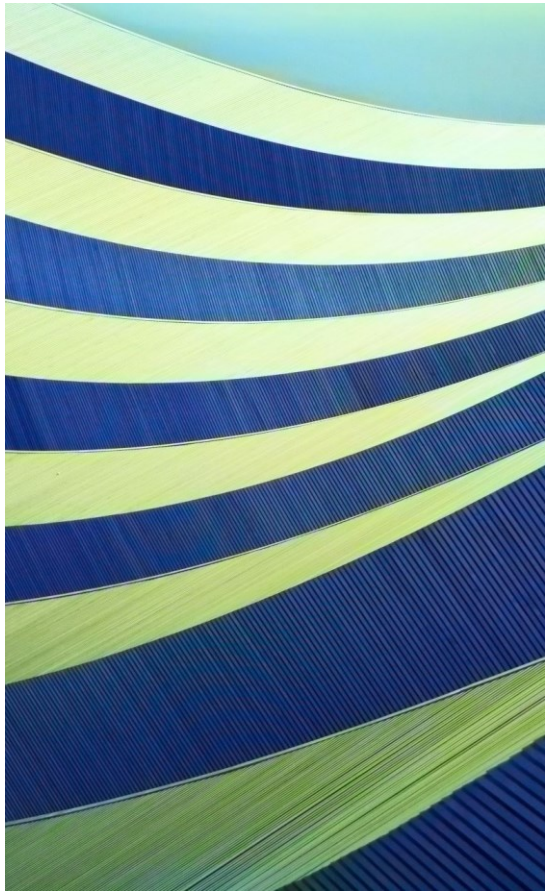
ENQ/DEQ Services to guarante transaction control over databases

Low-level data and memory access (e.g., dumpIt, outtrap, updateIOPL)

# BREXX MVS Integration Overview

BREXX includes extensions tailored to IBM MVS systems, adding powerful scripting capabilities for system interaction, dataset access, security, and console control.

- Access to datasets (PS, PDS), directory entries, record counts
- Use of SVCs and TSO console commands
- Low-level ENQ/DEQ locking mechanisms
- Dynamic privilege control and environment detection (Hercules, VM/370)
- Integration with NJE, SMF, RACF environments



If it can be called, piped, or poked—BREXX can handle it

# BREXX/370 Array Extensions

**BREXX vs. BREXX/370 – Variable Management**

**BREXX:**

- Variables in BREXX are organized using a **binary tree structure**.
- While flexible, this approach introduces **significant overhead** in:
  - **Memory usage** (due to metadata and pointer structures)
  - **Performance**, especially during **intensive iteration** or when dealing with a **large number of variables**
- Each variable access requires tree traversal, slowing down execution.

**BREXX/370:**

- Variables in BREXX are organized using a **binary tree structure**.
- Introduces **native array support**, allowing **direct indexed access** to data.
- Arrays eliminate the need for binary tree traversal, providing:
  - **Faster variable access**
  - **Reduced memory overhead**
  - Improved performance in **highly iterative or data-heavy tasks**

✅ **Key Advantage**: Arrays in BREXX/370 significantly optimize storage and execution speed, making it more efficient for large-scale or performance-critical REXX applications.

If I have time to press Enter, it's not fast enough.

# High-Performance Array Support in BREXX

BREXX introduces internal string, integer and float arrays, bypassing traditional REXX variable pools for faster data processing.

- Arrays are stored outside the normal symbol table / variable pool

- String array: `sarray[]`, with indexes via `sindex` and `sarrayhi[]`

- Integer and Float arrays: optimized for numeric lookup, manipulation, and batch updates

- Operations supported: sort, search, batch load/store

- Greatly reduces CPU load for large-scale operations vs. stem variables

- Designed for MVS workloads and system-level scripting efficiency

# sarray[]: High-Speed String Arrays

`sarray[]` is a fast, low-overhead string array structure designed for heavy-duty scripting.

- Defined as `char *sarray[sarraymax]`, outside normal REXX variable pools
- Directly accessed by index, avoiding stem variable performance limits
- `sarrayhi[]` tracks the highest index used in each array
- Optional `sindex[]` provides sorted views or lookup optimization
- Used for fast batch processing, logging, or system dataset parsing
- Ideal for MVS workloads where classic REXX is too slow

# sarray[]: Memory Management

- SCREATE – Create a new sarray instance
- SRESIZE – Resize an existing sarray
- SFREE – Free the memory associated with sarray
- SARRAY – Return info/status of an sarray

# sarray[]: Element Access & Assignment

- SSET – Set a value at a specific index
- SGET – Retrieve a value from a specific index
- SSUBSTR – Get a substring from an entry
- SWORD – Extract a word from an entry
- SLSTR – Convert list-style string into sarray

# Example: High-Speed sarray[] Access

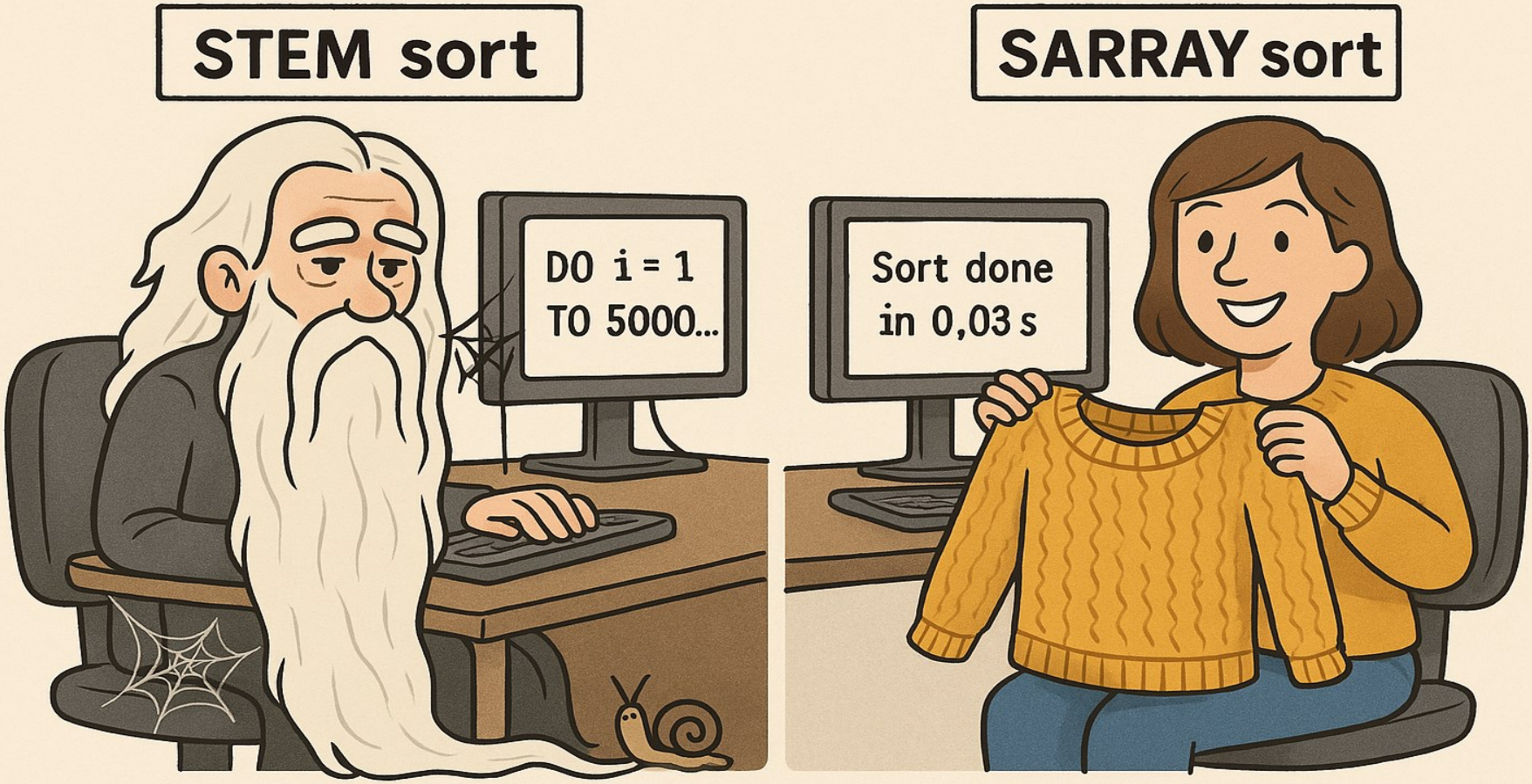
```
RFEEDIT  BREXX.V2R5M3.SAMPLES($SORT) - 1.00
Command ===> _
****** **********Autosave********************************** Top of Data *******
000001 smax=5000    /* Number of random entries */
000002 /* ---------------------------------------------------------------------
000003  * QuickSORT
000004  * ---------------------------------------------------------------------
000005  */
000006 srt=setupArray('QuickSort')
000007 call slist srt,1,10
000008 elp=time('e')
000009 call SQSORT srt,"QUICKSORT"
000010 elp=Trunc(time('e')-elp,3)
000011 say "quicksort  time "elp" seconds, items: "smax
000012 call slist srt,1,10
000013 /* ---------------------------------------------------------------------
000014  * ShellSORT
000015  * ---------------------------------------------------------------------
000016  */
000017 srt=setupArray('ShellSort')
000018 call slist srt,1,10
000019 elp=time('e')
000020 call SHSORT srt,"SHELLSORT"
000021 elp=Trunc(time('e')-elp,3)
000022 say "shellsort  time "elp" seconds, items: "smax
000023 call slist srt,1,10
000024 exit  _
000025 /* ---------------------------------------------------------------------
000026  * Some Subroutines
000027  * ---------------------------------------------------------------------
000028  */
000029 setupArray:
000030 parse arg stype
000031 srt=Screate(smax)
000032 do i=1 to smax
000033    call sset(srt,,right(RANDOM(0,10000),6,'0')' 'stype)
000034 end
000035 return srt
```

# Example: High-Speed sarray[] Access

```
     Entries of Source Array: 1
Entry   Data
-----------------------------------------------------
00001   002037 QuickSort
00002   007084 QuickSort
00003   001651 QuickSort
00004   002001 QuickSort
00005   006034 QuickSort
00006   009235 QuickSort
00007   005196 QuickSort
00008   000417 QuickSort
00009   003595 QuickSort
00010   004499 QuickSort
10 Entries
quicksort  time 0.121 seconds, items: 5000
     Entries of Source Array: 1
Entry   Data
---------------------------
00001   000004 QuickSort
00002   000005 QuickSort
00003   000006 QuickSort
00004   000008 QuickSort
00005   000010 QuickSort
00006   000010 QuickSort
00007   000010 QuickSort
00008   000011 QuickSort
00009   000015 QuickSort
00010   000016 QuickSort
10 Entries
```

```
     Entries of Source Array: 2
Entry   Data
-----------------------------------------------------
00001   009592 ShellSort
00002   000440 ShellSort
00003   003683 ShellSort
00004   006024 ShellSort
00005   004442 ShellSort
00006   003596 ShellSort
00007   000618 ShellSort
00008   004510 ShellSort
00009   000006 ShellSort
00010   008781 ShellSort
10 Entries
shellsort  time 0.306 seconds, items: 5000
     Entries of Source Array: 2
Entry   Data
-----------------------------------------------------
00001   000002 ShellSort
00002   000003 ShellSort
00003   000003 ShellSort
00004   000003 ShellSort
00005   000003 ShellSort
00006   000004 ShellSort
00007   000004 ShellSort
00008   000006 ShellSort
00009   000006 ShellSort
00010   000010 ShellSort
10 Entries
```

```
quicksort  time 5.655 seconds, items: 5000
shellsort  time 39.862 seconds, items: 5000
heapsort   time 12.749 seconds, items: 5000
Bubble Sort is not recommended for items > 250,
***
```




STEM sort: timeless in all the wrong ways.
SARRAY sort: so fast, you'll need a new hobby.

# sarray[]: Transformation & Modification

- SCHANGE – Replace occurrences of a substring
- SREVERSE – Reverse the order of entries
- SSWAP – Swap two entries by index
- SCLC – Clear array content
- SCOPY – Copy one sarray to another
- SINSERT – Insert a new entry
- SDEL – Delete an entry by index
- SPASTE – Append entries from another source

# sarray[]: Search & Selection

- SSEARCH – Search for an entry

- SSELECT – Select entries by pattern or value

- SKEEP – Keep matching entries

- SKEEPAND – Keep entries matching multiple conditions

- SDROP – Drop matching entries

# sarray[]: Sorting & Counting

- SQSORT – Quick sort for entries

- SHSORT – Shell sort (likely stable)

- SCOUNT – Count elements or pattern matches


The moment has come. It's showtime.
Destiny awaits.
Hercules Version : 4.5.0.10820-SDL-DEV-g5198616
Host name : eitri
Host OS : Linux-4.4.0-210-generic #242-Ubuntu SMP Fri Apr 16 09:57:56
Host Architecture : x86_64
Processors : MP=2
LPAR Name : HERCULES
Device number : 0:00C0
ZZZzz
The MVS 3.8j
Tur(n)key System
Update 08
TK3 created by Volker Bandke vbandke@bsp-gmbh.com
TK4- update by Juergen Winkelmann winkelmann@id.ethz.ch
see TK4-.CREDITS for complete credits

# sarray[]: Set Operations

- SMERGE – Merge sorted sarrays
- SINTERSECT – Intersection of arrays
- SDIFFERENCE – Difference (A not in B)
- SUNIFY – Remove duplicates

# sarray[]: Load & Store

- SREAD – Load sarray from file or source

- SWRITE – Write sarray to file or target

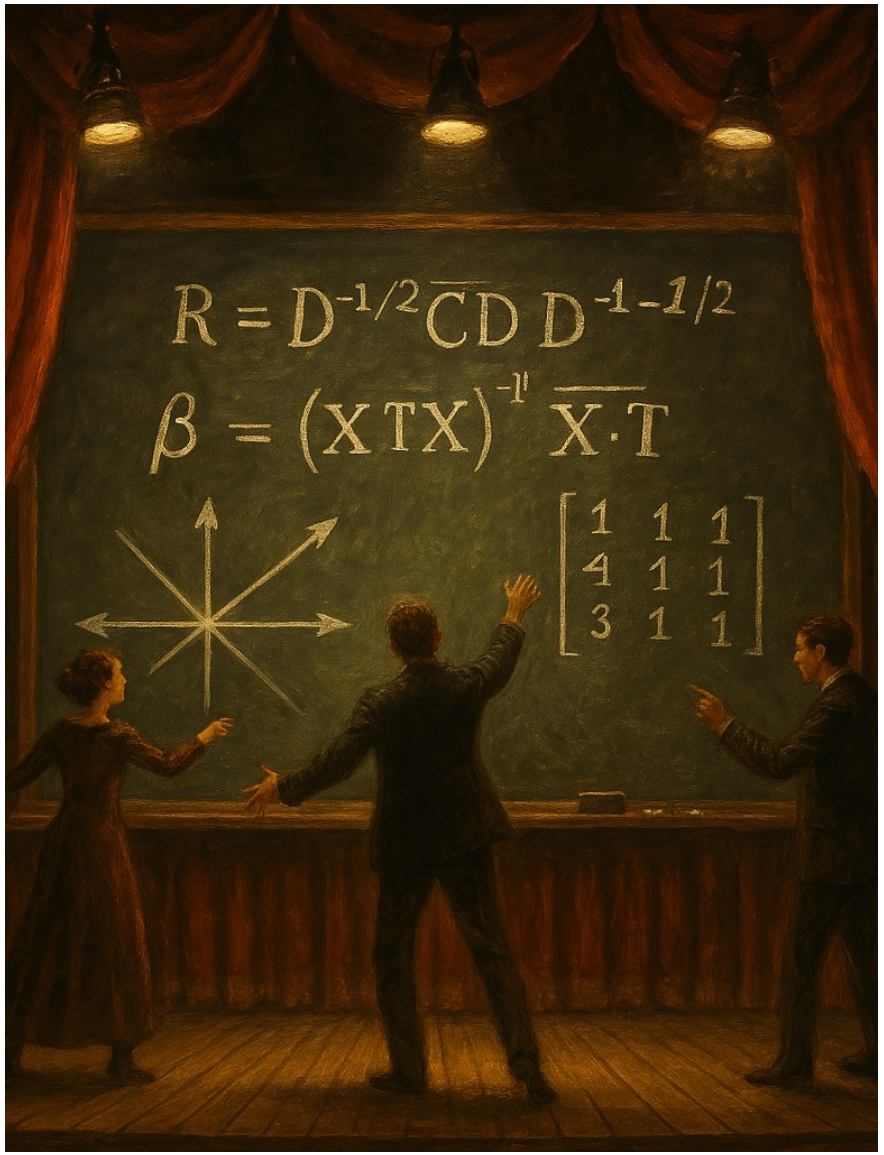
- S2LL – Convert to linked list

- S2IARRAY – Convert to integer array

- SLIST – Return sarray as a stem/list

Exploring the Unexpected Passion of Linear Algebra in BREXX

# I Came for the Vectors, Stayed for the Drama



# Why Matrix Operations Matter

**Matrix Operations** in BREXX are foundational tools for implementing many statistical and analytical methods:

**Correlation Analysis** – based on covariance and matrix algebra

**Regression Analysis** – uses matrix inversion and multiplication

**Factor Analysis** – relies on eigenvalue decomposition and matrix transformations

**Principal Component** Analysis (PCA) – transforms data into eigenvector space

**Data Normalisation**, aggregation, and reduction techniques

Enables advanced statistical modelling within REXX scripts


The moment has come. It's showtime.
Destiny awaits.
Hercules Version : 4.5.0.10820-SDL-DEV-g5198616
Host name : eitri
Host OS : Linux-4.4.0-210-generic #242-Ubuntu SMP Fri Apr 16 09:57:56
Host Architecture : x86_64
Processors : MP=2
LPAR Name : HERCULES
Device number : 0:00C0
ZZZzz
The MVS 3.8j
Tur(n)key System
Update 08
TK3 created by Volker Bandke vbandke@bsp-gmbh.com
TK4- update by Juergen Winkelmann winkelmann@id.ethz.ch
see TK4-.CREDITS for complete credits

# Matrix Operations: Creation & Access

MCREATE – Create a new matrix

MGET – Retrieve value at a given row/column

MSET – Set value at a given row/column

MDELCOL – Delete a column

MDELROW – Delete a row

# Matrix Operations: Manipulation

- MCOPY – Copy one matrix to another
- MTRANSPOSE – Transpose matrix (flip rows/columns)
- MNORMALISE – Normalize data across rows or columns
- MINVERT – Invert the matrix (if square and invertible)

# Matrix Operations: Arithmetic

- MMULTIPLY – Multiply two matrices
- MADD – Add two matrices
- MSUBTRACT – Subtract matrices
- MPROD – Element-wise product
- MSQR – Square each element
- MSCALAR – Multiply matrix by a scalar

# Linear Algebra:

# Correlation & Regression

Mathematical formulas using matrix operations

$$R = \left(\sqrt{D}\right)^{-1} C \left(\sqrt{D}\right)^{-1}$$

where: $C$ = Covariance matrix

$D$ = Diagonal matrix of variances

$D = \text{diag}(\text{Var}(X))$

Linear regression coefficients $\beta$:

$$\beta = (X^T X)^{-1} X^T$$

where: $X$ = Matrix of input variables (predictors)

$Y$ = Output vector (target variable)

$\beta$ = vector of regression coefficients

And now, before we're escorted out by an angry mob with torches, we shall summarise… and leg it.

# BREXX/370 — Summary

**Rooted in Proven Design**
Built upon the original BREXX interpreter by Vasilis Vlachoudis — a compact, portable implementation of REXX that set the foundation.

**Adapted for Mainframe Environments**
BREXX/370 brings this design to the IBM MVS 3.8 platform, extending core functionality while remaining true to its architectural simplicity.

**System-Level Integration**
Deep integration with MVS subsystems: VSAM, NJE38, formatted screen I/O, and more — enabling seamless scripting for complex system interactions.

**Expanded Capabilities**
Enhanced with array support, embedded functions, and matrix operations — enabling data analysis, transformation, and advanced scripting.

**Sustainable and Documented**
Continuously maintained, with full documentation, source availability, and a transparent development approach.

**Same Language, New Ground**
BREXX/370 continues the REXX tradition: readable, robust, and remarkably adaptable — now fully at home on the mainframe.

# The BREXX/370 Development Team

Mike & Peter

REXX reads like English, runs like C, and debugs like a dream

So long, and thanks for all the STEMs!

From DOS to MVS — BREXX never stopped listening

```
NAME
       brexx370-stubbornly portable, surprisingly powerful
       REXX interpreter for IBM maintrames

SYNOPSIS
       brexx370 [--retro] [---nostalgia] [--macro=love]
       script.rexx

DESCRIPTION
       BREXX/370 brings the minimalist elegance of BREXX to
       the world of MVS 3.8 and beyond.
       It interprets REXX scripts with speed, style, and
       suspiciously few lines of node.
       Originally designed for platforms no one admits using
       anymore, it now runs on systems your grandchildren
       won't believe ever existed.

FEATURES
       • Inline compilation using _CodeAddByte() – because
         we could
       • Controi flow patching via CODEFIXUP macros –
         trust the process
       • Parses like it's 1889, runs like it's 2025
       • No AST. No regrets.

BUGS   • May cause feelings of nostalgia
       • Cannot compite your life decisions
       • Occasionally too efficient for its own good

AUTHOR Written by people who looked at IBM's architecture
```