



# Rexx and NetRexx

Warpstock 2017, Rotterdam

René Vincent Jansen 2017-05-21

# Agenda

- ✦ REXX
  - ✦ What is it
  - ✦ Compact History
  - ✦ Where are we now?
  - ✦ Future as open source



# Rexx



- ✦ Scripting Language
- ✦ Started out as an interpreter
- ✦ ‘Human instead of computer oriented’
- ✦ Family of implementations

# What is a scripting language

- A **scripting language**, **script language** or **extension language** is a programming language that allows control of one or more software applications. "Scripts" are distinct from the core code of the application, as they are usually written in a different language and are often created or at least modified by the end-user. Scripts are often interpreted from source code or bytecode, whereas the applications they control are traditionally compiled to native machine code.
- Early script languages were often called batch languages or *job control languages*. Such early scripting languages were created to shorten the traditional edit-compile-link-run process.

# History



- ✦ 1979, Mike Cowlishaw, IBM VM, Hursley
- ✦ Initially reprimanded, but strong undercurrent and eventually Mike was made an IBM fellow
- ✦ Was called REX (Reformed eXecutor), but after IBM lawyers intervened and money was spent, REXX was the new name
- ✦ *Bacronym* of Restructured Extended Executor

# Roots - EXEC2

- ✦ Rexx' predecessor was called EXEC2
  - ✦ A *Single* macro language for many applications (Stephenson, 1973)
- ✦ This language consisted mainly of Ampersands
  - ✦ `&IF &NODE&J I= &LOCAL &USER = &STRING OF`
- ✦ The plan was to have a PL/1 based replacement that 'turned things around' - a small percentage would be literals

# What came out

- ✦ A language designed for the user (that is, programmer) instead of the language implementer
  - ✦ Unlimited precision arithmetic
  - ✦ Strong PARSE statement
  - ✦ Nothing to declare
  - ✦ Strong built-in TRACE tool
  - ✦ Relatively few keywords
  - ✦ Strong String processing

# History



- ✦ Strong user community influence, VMNET was new
- ✦ Principle of design and document first, then implement and test
- ✦ Circulated on the network between IBM sites, made into a product by customer demand

# History (IBM)

- ✦ 1979 Mike Cowlshaw starts work on REX
- ✦ 1981 First Share presentation on REX
- ✦ 1982 First non-IBM location to get REX is SLAC
  - ✦ (Stanford Linear Accelerator Center)
- ✦ 1983 Command Language for IBM VM/CMS
- ✦ 1989 REXX Supported on MVS (TSO Extensions V2)
- ✦ 1990 REXX Supported on **OS/2** (EE) and OS/400



# REX Reference Summary

VM/370 • CMS

First Edition (November 1980) - for REX version 2.08

## CONTENTS

REX EXEC	1
Data items	2
Expressions	2
Statements	3
Templates and Parsing	5
Compound variable names	5
Built-in variables	5
Issuing commands to CMS	5
Interactive Debugging	6
REXFNS function package	6
REXFNS2 function package	7
REXWORDS function package	7
Utility Modules	7
Restrictions	8
The Command and Exec Plist	8
Sample REX Exec.	9

REX is a command programming language which allows you to combine useful sequences of commands to create new commands. It is used in conjunction with, or as a replacement for, the CMS EXEC and EXEC 2 languages. REX is especially suitable for writing Execs or editor macros, but is also a useful tool for algorithm development.

## REX EXEC

- Invoke using: REX [*parameter*]
- REX ? describes REX EXEC and how to use the on-line help and tutorial.
- REX I installs REX in your CMS system under the name EXEC so that you may execute Execs written in any of the three languages. You probably will want to put 'EXEC REX I' in your PROFILE EXEC.
- REX enters the on-line help, viewing an index of topics.
- REX 200nn (where nn is a REX error message number) describes the meaning of the error message and the likely cause of the error.
- REX kkk (where kkk is a REX keyword) gives immediate information on the specified topic.

### Key to notation used on this card:

- GOTHIC — indicates language keywords
- italics — indicate defined syntactic units
- [ ] — brackets indicate an optional item
- ... — ellipses mean multiple items are allowed
- { } — braces specify list of alternatives (choose one)
- | — separates alternatives in a list

IBM INTERNAL USE ONLY

## DATA ITEMS

The REX language is designed for the easy manipulation of character strings. Its expressions and instructions manipulate the following items:

- string** A string is a quoted string of characters. Use two quotes to obtain one quote inside the string. The string may be specified in hexadecimal if the final quote is followed by an X. Some valid strings are: "Next 'Don't touch' '1de8'x"
- number** A number is a string of up to 9 decimal digits before and/or after an [optional] decimal point. It may have a leading sign. Some valid numbers are: 17 98.07 .101
- name** A name refers to a variable, which can be assigned any value. It may consist of up to 150 characters from the following selection: A-Z, a-z, 0-9, @ # \$ % & \_ . ? ! £ The first character may not be a digit or period, except if the name consists of the period alone. The name is translated to upper case before use, and forms the initial value of the variable. Some valid names are: Fred COST? next\_index A.j
- function-call** A function-call invokes an external routine with 0 to 7 arguments. The called routine will return a character string. A function-call has the format:  
function-name([*expr*] [*expr*]...)  
Function-name must be adjacent to the left parenthesis, and may be a name or a string.

## EXPRESSIONS

Most REX statements permit the use of expressions, following the style of PL/I. Expressions are evaluated from left to right, modified by the priority of the operators (as ordered below). Parentheses may be used to change the order of evaluation. All operations (except prefix operations) act on two items, and result in a character string.

- Prefix + - ~ Prefix operations: Plus; Minus; and Not. (For + and -, item must evaluate to a number, for ~, it must be '1' or '0'.)
- \* / // Multiply; Divide; Divide and return the remainder. (Both items must evaluate to numbers.)
- + - Add; Subtract. (Both items must evaluate to numbers.)
- (blank) || Concatenate: with or without blank. Abutal of items causes direct concatenation.
- = == > < > < > < > Comparisons (arithmetic compare if both items evaluate to a number). The == operator checks for an exact match.
- & Logical And. (Both items must be '0' or '1'.)
- | && Logical Or; logical Exclusive Or. (Both items must be '0' or '1'.)

The results of arithmetic operations are expressed to the same decimal precision as the more precise of the two items. For example, 123.57 + 12 will result in 135.57. Results of division are rounded rather than truncated.

IBM INTERNAL USE ONLY

## STATEMENTS

REX statements are built out of clauses consisting of a series of items, operators, etc. The semicolon at the end of each clause is often not required, being implied by line-ends and after the keywords THEN, ELSE, or OTHERWISE. A clause may be continued from one line to the next by using a comma at the end of the line. This then acts like a blank. Open strings or comments are not affected by line ends, and do not require a continuation character.

Keywords are shown in capitals in this list, however they may appear in either (or mixed) case. Keywords are only reserved when they are found in the correct context.

In the descriptions below: *expr* is an expression as described above; *stmt* is any one of the listed statements; *template* is a parsing template, as described in a later section; *name* is usually the name of a variable (see above).

- name = [expr];** assignment: the variable *name* is set to the value of *expr*.
- expr;** the value of *expr* is issued as a command.
- ADDRESS [{name}|string] [expr];** redirect commands or a single command to new environment.
- ARGS [template];** parse the argument string into variables. The contents of all variables except the last are translated to upper case. (Note: the first argument is always the name of the Exec or subroutine.)
- CALL name [expr];** call an internal subroutine. On return, the variable *name* will have the value from the RETURN statement. Subroutines may be called recursively.
- DO [name = expr] [TO expr] [BY expr] [{UNTIL|WHILE} expr];** statement grouping with optional repetition and condition. The variable *name* is stepped from *expr1* to *expr2* in steps of *expr3*. These *exprs* are evaluated only once at the top of the loop and must result in a whole number. This iterative phrase may be replaced by a single *expr* which is a loop count (no variable used). If a WHILE or UNTIL is given, its *expr* must evaluate to '0' or '1'. The condition is tested at the top of the loop if WHILE or at the bottom if UNTIL.
- DROP [name]...;** drop (reset) the named, or all, variables.
- EXIT [expr];** leave the Exec [with return code].
- IF expr {;|THEN} stmt [ELSE[;] stmt]** if *expr* evaluates to '1', execute the statement following the ';' or THEN. Otherwise (evaluates to '0') skip that statement and execute the one following the ELSE clause, if present.
- INTERPRET expr;** evaluate *expr* and then execute the resultant string as if part of the original program.
- ITERATE [name];** start next iteration of innermost repetitive loop [or loop with control variable *name*].
- LEAVE [name];** terminate innermost loop [or the loop with control variable *name*].
- NOP;** dummy statement, has no side-effects.
- PARSE ARGS [template];** ARGS without upper case translation.
- PARSE PULL [template];** PULL without upper case translation.

IBM INTERNAL USE ONLY

## PARSER

- PARSE SOURCE [template];** parse program source description 'CMS {COMMAND|FUNCTION} fn ft fm'.
- PARSE VAR name [template];** parse the value of *name*.
- PARSE VERSION [template];** parse data describing interpreter.
- PROCEDURE;** start a new generation of variables within a subroutine.
- PULL [template];** read the next string from the system queue ("stack") and parse it into variables. The contents of all variables except the last are translated to upper case.
- PUSH [expr];** push *expr* onto head of the system queue ("stack LIFO").
- QUEUE [expr];** add *expr* to the tail of the system queue ("stack FIFO").
- RETURN [expr];** evaluate *expr* and return the value to the caller. (Pushes the value onto the system queue if not a function or internal call.)
- SAY [expr];** evaluate *expr* and then display the result on the user's console, using current line size.
- SELECT;**  
[WHEN *expr*{;|THEN} *stmt*]...  
[OTHERWISE[;] *stmt*]...  
**END;** the WHEN *exprs* are evaluated in sequence until one results in '1'. the *stmt* immediately following it is executed and control then leaves the construct. If no *expr* evaluates to '1', control passes to those *stmts* following the OTHERWISE which must then be present.
- SIGNAL {ON|OFF} {name}|string;** enable or disable exception traps. (The condition must be ERROR, EXIT, NOVALUE, or SYNTAX, and control will pass to the label of that name should the event occur while ON.)
- SIGNAL expr;** go to the label specified. Any pending statements, DO ... END, INTERPRET, etc. are terminated.
- TRACE expr;** if numeric then (if negative) inhibit tracing for a number of clauses, or (if positive) inhibit debug mode for a number of clauses. Otherwise trace according to first character of the value of *expr*:  
'E' = trace after non-zero return codes.  
'C' = trace all commands.  
'A' = trace all clauses.  
'R' = trace all clauses and expressions.  
'I' = as 'R', but trace intermediate evaluation results and name substitutions also.  
'L' = trace only labels.  
'S' = display rest of program without any execution (shows control nesting).  
'0' or null = no trace.  
'!' = trace according to the next character, and inhibit command execution.  
'?' = turn debug mode (pause after trace) on or off.
- name:** form of labels for CALL or SIGNAL. The colon always acts as a clause separator.
- /\* form of comment \*/** may be used anywhere except in the middle of a name or string. (Required on first line to identify REX Execs.)

IBM INTERNAL USE ONLY

## TEMPLATES for ARGS, PULL, and PARSE

The PULL, ARGS, and PARSE instructions use a *template* to parse a string. The template specifies the names of variables that are to be given new values, together with optional triggers to control the parsing. Each *name* in the template is assigned one word (without any leading or trailing blanks) from the input string in sequence, except that the last *name* is assigned the remainder of the string (if any) unedited. If there are fewer words in the string than names in the template, all excess variables are set to null. In all cases, all the variables in the template are given a new value.

If PULL or ARGS are used, then the separately assigned words only will first be translated to upper case. When this translation is not desired, use the PARSE instruction.

The parsing algorithm also allows some pattern matching, in which you may "trigger" on either a string or a special-character (the '(' is useful in the CMS environment, for example). A special-character is one of:

+ \* / | & = > < > ; : ) (

If the template contains such a trigger, then alignment will occur at the next point where the trigger exactly matches the data. A trigger match splits the string up into separate parts, each of which is parsed in the same way as a complete string is when no triggers are used.

## COMPOUND VARIABLE NAMES

Any name may be "compound" in that it may be composed of several parts (separated by periods) some of which may have variable values. The parts are then substituted independently, to generate a fully resolved name. In general

$s_0.s_1.s_2. \dots .s_n$  will be substituted to form:  
 $d_0.v_1.v_2. \dots .v_n$  where  $d_0$  is upper case of  $s_0$   
 $v_1-v_n$  are values of  $s_1-s_n$ .

This facility may be used for content-addressable arrays and other indirect addressing modes. As an example, the sequence:

J = 5; A.J = 'fred';

would assign 'fred' to the variable 'A.5'.

## BUILT-IN VARIABLES

There are two built-in variables:

- RC is set to the return code after each executed command.
- SIGL is set to the line number of last line that caused SIGNAL, CALL or RETURN jump.

## ISSUING COMMANDS to CMS

The default environment for commands in Execs is CMS. A command is an expression, which may include function-calls, arithmetic operations, and so on. Operators or other special characters (for example '(' or '\*') must therefore be specified in a string if they are to appear in the issued command.

To issue a CP command or call another Exec, the first word of the expression value should be 'CP' or 'EXEC' respectively. Use the OBEY command instead if full CMS command resolution is to be applied.

In editor macros, the default environment for commands is the same as the filetype of the macro.

IBM INTERNAL USE ONLY

# 1980

# vm/370 - CMS

# History

- ✦ 1985 Mansfield REXX for PC/DOS
- ✦ 1987 AREXX for the Commodore Amiga
- ✦ 1990 Uni-Rexx for Unix, AIX, Tandem (Kilowatt)
- ✦ 1991 REXX for DEC/VMS
- ✦ 1992 REXX/imc and Regina

# Regina

- ✦ By Anders Christensen, maintained by Mark Hessling
- ✦ For Linux, FreeBSD, Solaris, AIX, HP-UX, etc.) and also to OS/2, eCS, DOS, Win9x/Me/NT/2k/XP, Amiga, AROS, QNX4.x, QNX6.x, BeOS, MacOS X, EPOC32, AtheOS, OpenVMS, SkyOS and z/OS OpenEdition.

# History

- ✦ 1990 First Rexx Language Association Symposium Held
- ✦ 1991 First REXX ANSI Committee meeting held
- ✦ 1996 ANSI X3274–1996 “Information Technology – Programming Language REXX”.

# History

- ✦ 1989 Rexx is a now also a compiled language
  - ✦ Based on research by IBM Haifa, developed by IBM Vienna Software Development Lab, Austria
  - ✦ runtime performance improvements, sourceless distribution of scripts and programs

# Object Rexx

- ✦ Simon Nash, Hursley Lab, starting 1989
- ✦ Long gestation process - delivered first in OS/2 in 1997
- ✦ Commercial versions for Windows NT and AIX
- ✦ In 2004 released as Open Source by IBM and controlled by The Rexx Language Association
- ✦ Henceforth known as Open Object Rexx (ooRexx)
- ✦ [www.oorexx.org](http://www.oorexx.org)



# NetRexx

- ✦ The other Object Oriented successor to Classic Rexx
- ✦ 1995, Mike Cowlishaw
- ✦ Runs on the Java VM
- ✦ Compiles NetRexx to Java classes
- ✦ Added an interpreter in 2000
- ✦ Will be open sourced, probably this year



# Where are we now

- ✦ Rexx had its heyday of new activity around 1995
- ✦ Reputation went -undeservedly- downhill with the “demise” of OS/2 and Workplace OS
- ✦ Classic Rexx is Strong on the Mainframe
- ✦ For the open source implementations, life has just begun
- ✦ There is more value there than in most places

# Wikipedia

## IBM NetRexx

From Wikipedia, the free encyclopedia

**NetRexx** is [IBM](#)'s implementation of the [Rexx programming language](#) to run on the [Java virtual machine](#). It supports a classic Rexx syntax along with considerable additions to support [Object-oriented programming](#) in a manner compatible with Java's [object model](#). The syntax and object model differ considerably from [Open Object Rexx](#), another IBM object-oriented variant of Rexx which has been released as [open source software](#).

NetRexx is free to download from [IBM\[1\]](#).

# Peculiarities of NetRexx

- The Rexx Data type - implicit in other implementations, but not named due to untyped usage
- PARSE
- TRACE
- Arbitrary numeric precision & decimal arithmetic
- Concatenation by abuttal
- No reserved words
- Case insensitive
- Automates type selection and declaration
- Autogeneration of JavaBeans properties (with *properties indirect*)
  
- Philosophy: a language should be easy for users, not interpreter writers
- 'No reserved words' ensures that old programs are never broken by language evolution

# The Rexx Data Type

- This is where statically typechecked meets type-less
- A Rexx instance can be a number or a (Unicode) string of characters
- $3+4$  is the same as "3" + "4"
- We can perform (arbitrary precision) arithmetic on it when it is a number
- The Rexx type keeps two representations under the covers (if needed)
  
- The Rexx way to handle decimal arithmetic ended up in Java and in IEEE 754r, implementation of **BigDecimal** actually written in NetRexx
  
- Automation inter-conversion with Java String class, char and char[] arrays, and numeric primitives (optional)

You can forego the language and use the Rexx Datatype, from the runtime package, in your Java source

Equally, you can decide not to use the Rexx type at all in your NetRexx source

# Numeric Precision

(**options binary** to avoid this and have Java primitive types as much as possible)

Rexx has arbitrary precision numbers as a standard - implemented in the runtime Rexx Datatype.

say 1/7

**numeric digits 300**

```
0.14285714285714285714285714285714285714
2857142857142857142857142857142857142857
1428571428571428571428571428571428571428
5714285714285714285714285714285714285714
2857142857142857142857142857142857142857
1428571428571428571428571428571428571428
5714285714285714285714285714285714285714
2857142857142857142857142857142857142857
142857142857142857142857142857142857
```

# Parse

- not your standard regexp parser - it is template based
- can do lispy things

```
cdr = "foo bar baz"  
loop while cdr <> "  
    parse cdr car ' ' cdr  
    line = line car.upper(1,1)  
end
```

Would be this  
in NetRexx



# Built-in TRACE

- for you (and me) who still debug best using print statements - saves time
- adds them automatically during compile
- can leave them in and switch off during runtime
- best way to debug server type software
- can 'trace var' to keep a watchlist
- or 'trace results' to see results of expressions

```
class fact
  method main(args=String[]) static
    factorial(5)

  method factorial(number) static
    trace results
    if number = 0 then return 1
    else return number * factorial(number-1)
  ...
```

```
--- fact.nrx
8 *==* if number = 0
>>> "0"
9 *==* else
*==*   return number * factorial(number-1)
>>> "4"
8 *==* if number = 0
>>> "0"
9 *==* else
*==*   return number * factorial(number-1)
>>> "3"
8 *==* if number = 0
>>> "0"
9 *==* else
*==*   return number * factorial(number-1)
>>> "2"
8 *==* if number = 0
>>> "0"
9 *==* else
*==*   return number * factorial(number-1)
>>> "1"
8 *==* if number = 0
>>> "0"
9 *==* else
*==*   return number * factorial(number-1)
>>> "0"
8 *==* if number = 0
>>> "1"
*==*       then
*==*       return 1
>>> "1"
>>> "1"
>>> "2"
>>> "6"
>>> "24"
>>> "120"
```

# Automated Type Selection and Declaration

```
package com.abnamro.midms.util.ssl
import java.io.
import java.net.
import java.rmi.server.
import javax.net.ssl.
import java.security.KeyStore
import javax.security.cert.X509Certificate
-- mind: if this does not compile you probably do not have jsse.jar on your classpath
class RMISSSLServerSocketFactory implements RMIServerSocketFactory, Serializable

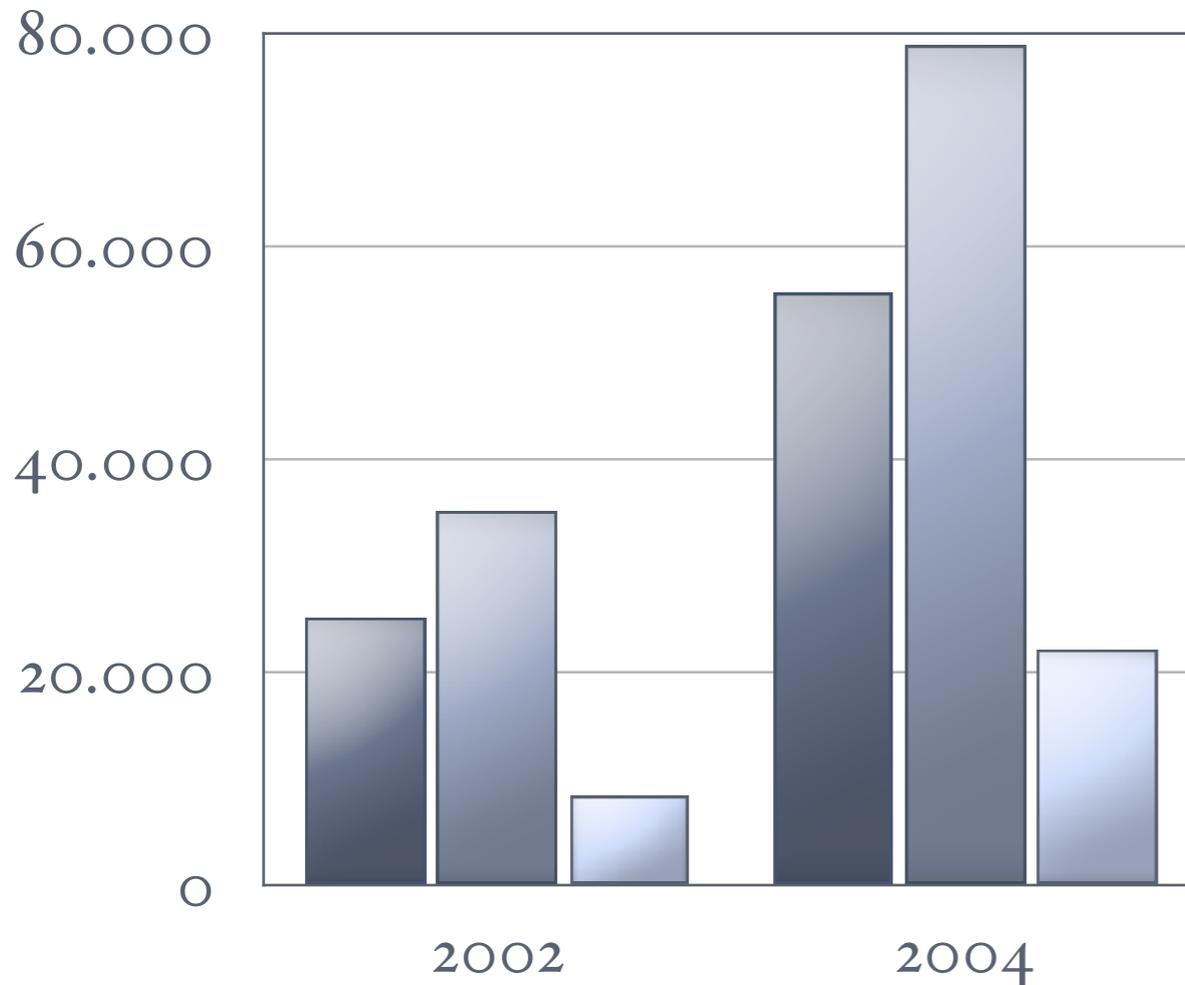
method createServerSocket(port=int) returns ServerSocket signals IOException
do
  -- set up key manager to do server authentication
  passphrase = char[] "passphrase".toCharArray()

  ctx      = SSLContext.getInstance("TLS")
  kmf      = KeyManagerFactory.getInstance("SunX509")
  ks       = KeyStore.getInstance("JKS")

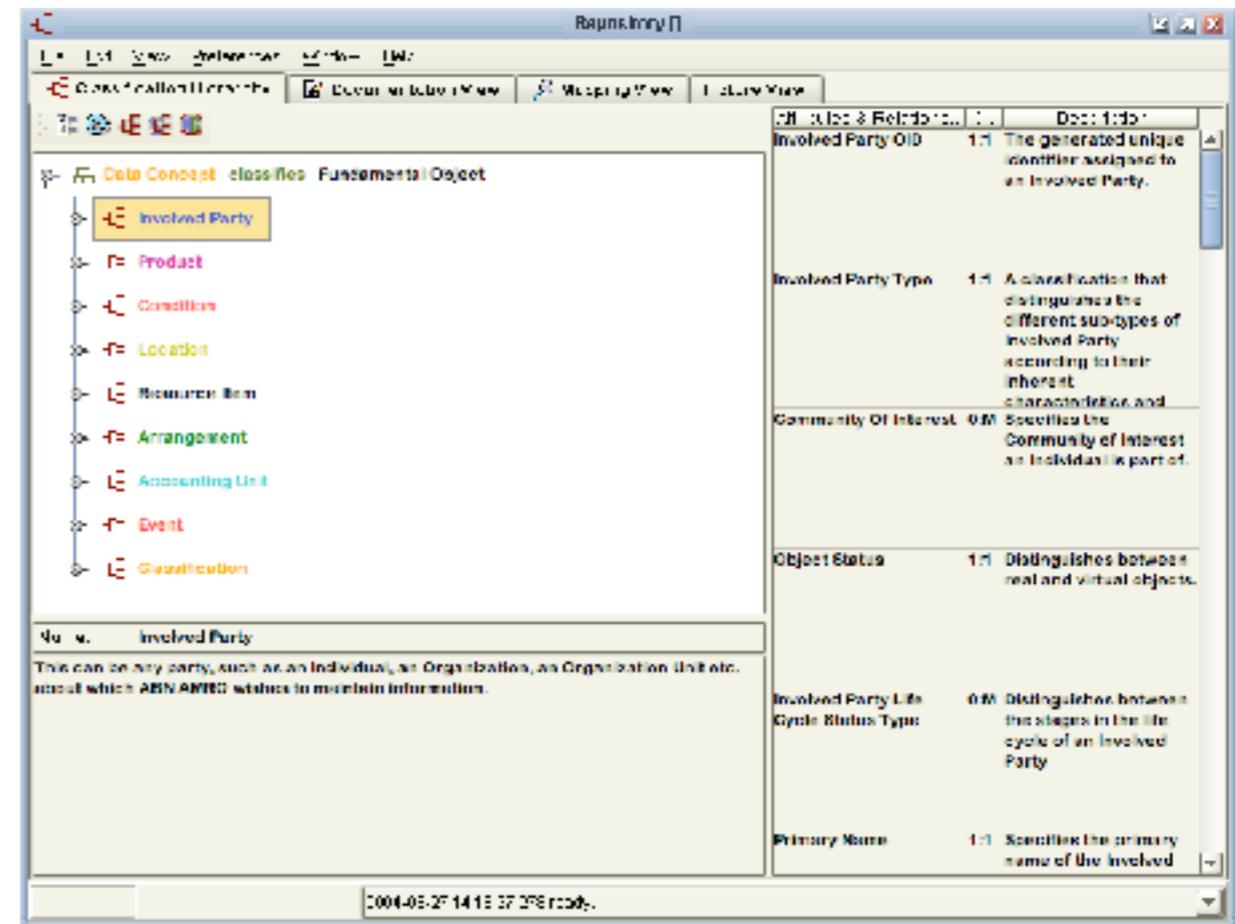
  ks.load(ClassLoader.getSystemClassLoader().getResourceAsStream("dmskey"), passphrase)
  kmf.init(ks, passphrase)
  ctx.init(kmf.getKeyManagers(), null, null)
  ssf = ctx.getServerSocketFactory()

catch e=Exception
  e.printStackTrace()
end
return ssf.createServerSocket(port)
```

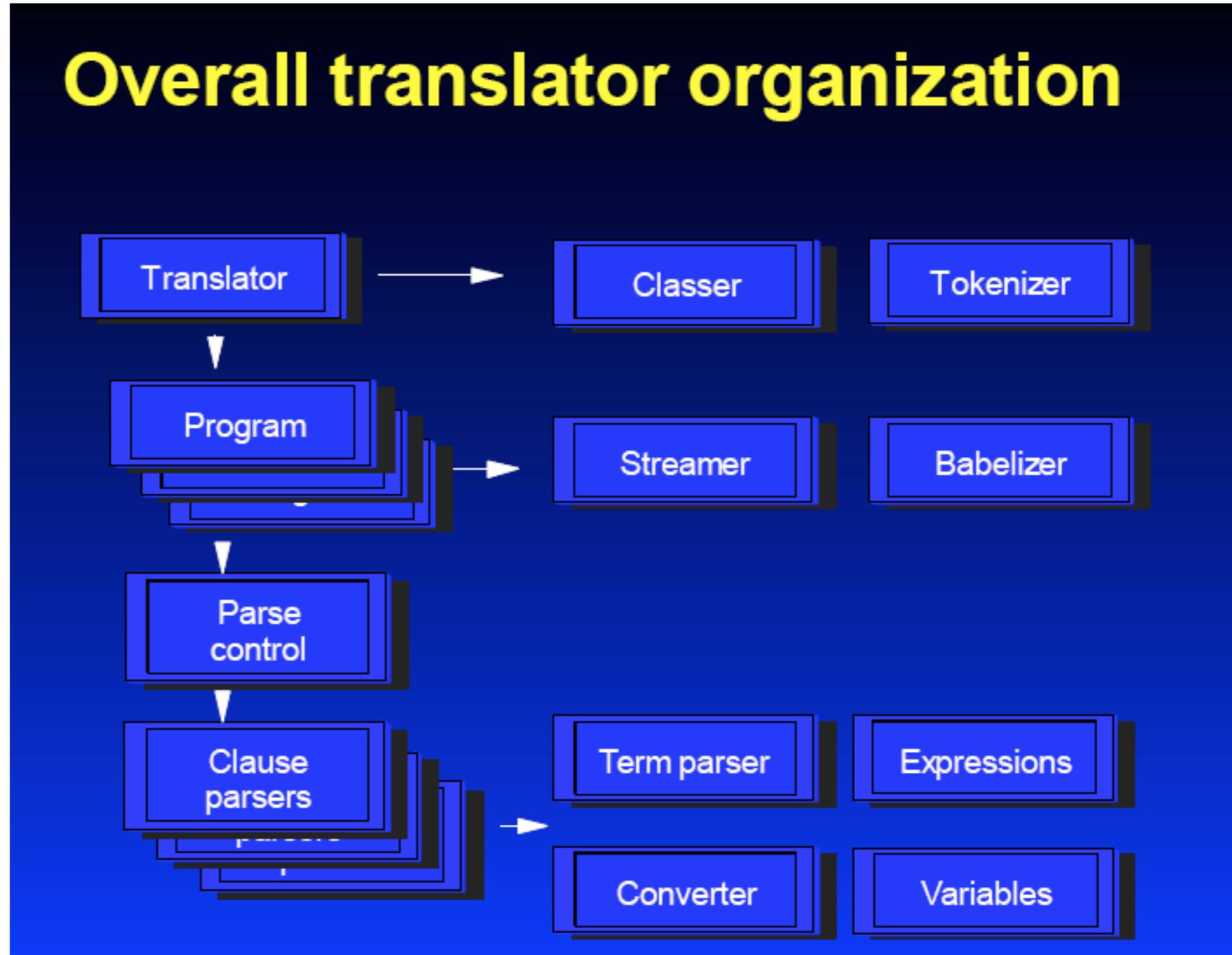
# Saving ±40% of lexical tokens in your source



- NetRexx Sourcelines
- NetRexx Generated Java
- NetBeans Generated Java



# Translation



# Parsing

- No upfront parsing - handwritten lexer & parser combo does 'on demand' parsing
- Parse on a 'Clause' base
- Stops quickly after errors in all three phases
  - Clear error messages, 'land on them' in IDE's (or Emacs in my case)

# Calling the compiler

- Finding the compiler: here it is a fact of life that sometimes this sits in tools.jar, sometimes rt.jar, sometimes classes.jar
- We are doing some searching for the usual suspects, but for some platforms, in the end, the user needs to know where it is and put it on the classpath
- Since some years ago we include the ecj (eclipse) compiler
- can use alternative compilers, jikes (is that still around?), or ibm jvm compilers

## Use in *scripting* mode

- Compiler adds boilerplate when needed
- and leaves it out when already there

```
say "hello JVM languages Summit!"
```

generates:

```
/* Generated from 'hello.nrx' 22 Sep 2008 19:57:00 [v3.00] */  
/* Options: Crossref Decimal Format Java Logo Replace Trace2 Verbose3 */  
  
public class hello{  
  private static final java.lang.String $0="hello.nrx";  
  
  public static void main(java.lang.String $0s[]){  
    netrexx.lang.RexxIO.Say("hello JVM languages Summit!");  
    return;}  
  
  private hello(){return;}  
}
```

# Runtime

NetRexxR.jar is currently 45463 bytes

## Contains

the **Rexx** datatype

console I/O like **say** and **ask**

Some Exceptions like `BadNumericException` - consequence of calling number methods on Rexx strings

Support for **Trace**

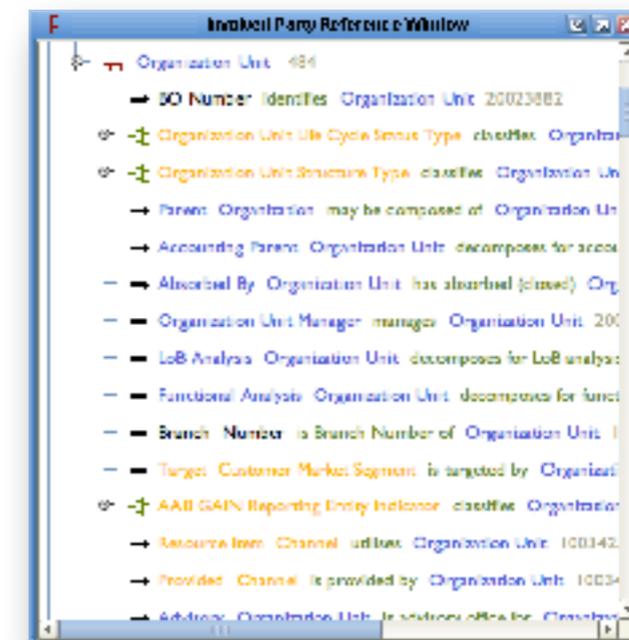
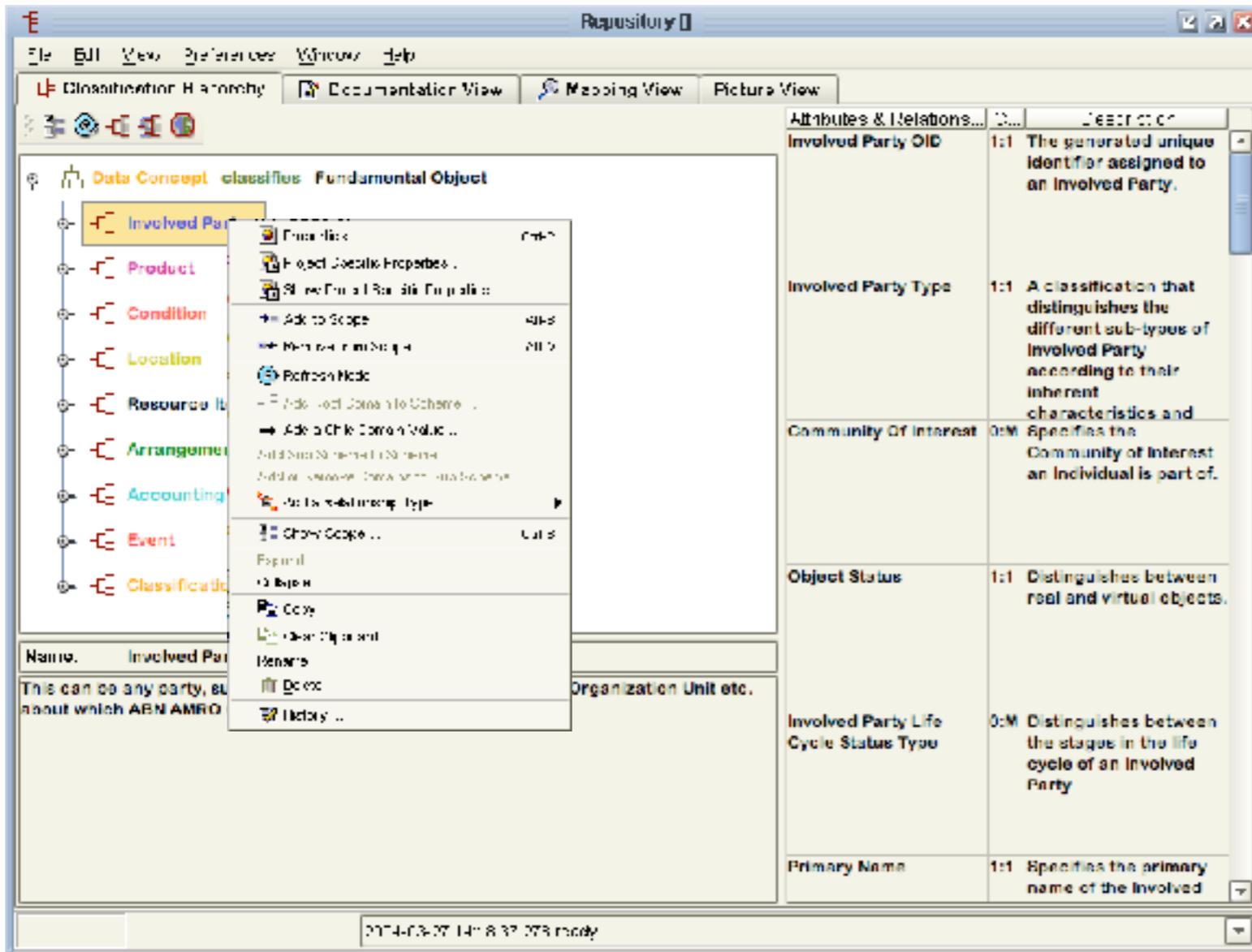
Support for **Parse**

**This is still a reasonable size for applet support**

## Easy integration with all existing Java infra

- Successfully and easily use
  - Java Collection Classes
  - NetBeans
  - Antlr
  - Hibernate
  - JSF
  
  - You name it.

# Swing GUIs using NetBeans



# Antlr - specifying the interpreter for a DSL (called *bint*)

## bint.g

```

header
{
    package com.abnamro.midms.bint;
}

{
    import java.io.*;
    import java.rmi.*;
}

class bintParser extends Parser;
options {
    k = 2;
    exportVocab=bint;
    codeGenMakeSwitchThreshold = 2;
    codeGenBitsetTestThreshold = 3;
    buildAST = false;
}

{
    public bintInterface bintInstance ;
}

program
    : (statement)+ EOF
    ;

statement

```

## bint.nrx

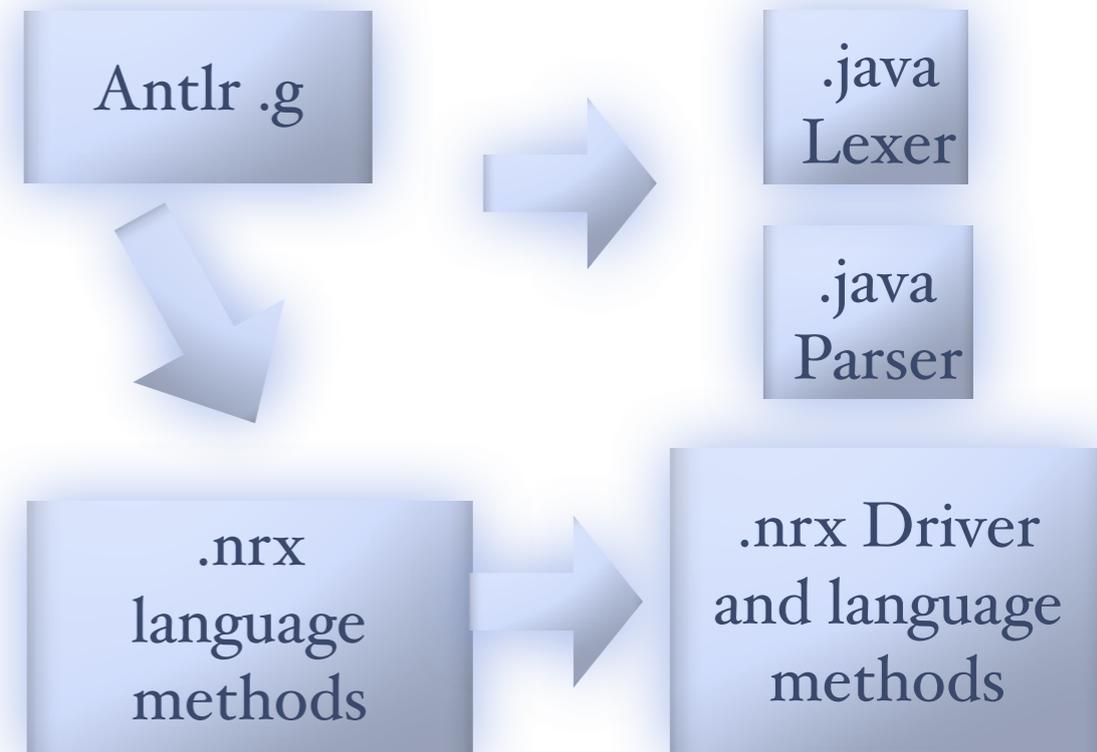
```

class bint implements bintInterface

method parseFile(s=InputStream)
do
    lexer = bintLexer(s)
    parser = bintParser(lexer)
    parser.bintInstance = this;
    parser.program()

catch e = Exception
    System.err.println("parser exception: " e)
    e.printStackTrace()
    errorText = e.getMessage()
    this.rollbackAndQuit()
end -- do
return

```



# Java Server Faces

```

<f:view>
<h:form id="AddSystemForm">

    <h:inputText id="PrimaryNameIn" value="#{addSystem.primaryName}" />
<b> <h:outputText id="PrimaryNameOut" value="System Name" /></b><br>

    <h:inputText id="UriIn" value="#{addSystem.systemAddress}" />
<b> <h:outputText id="UriOut" value="IP Address or DNS Name" /></b>

    <h:commandButton action="#{addSystem.add}" value="Add System" /><br><br>
    <h:commandButton action="#{addSystem.goBack}" value="Return to Subscription" /><br>
</h:form>
</f:view>

```



```

AddSystem.mrx: /Volumes/Workspace/src/com/abnamro/urd/admin/AddSystem.mrx
method setSystemAddress(s:String)
    this.systemAddress = s

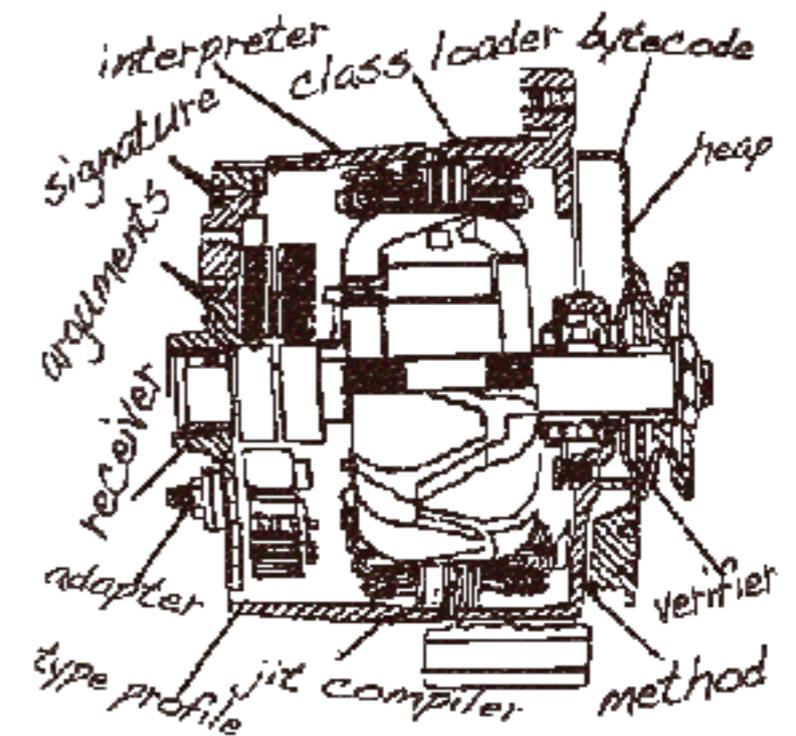
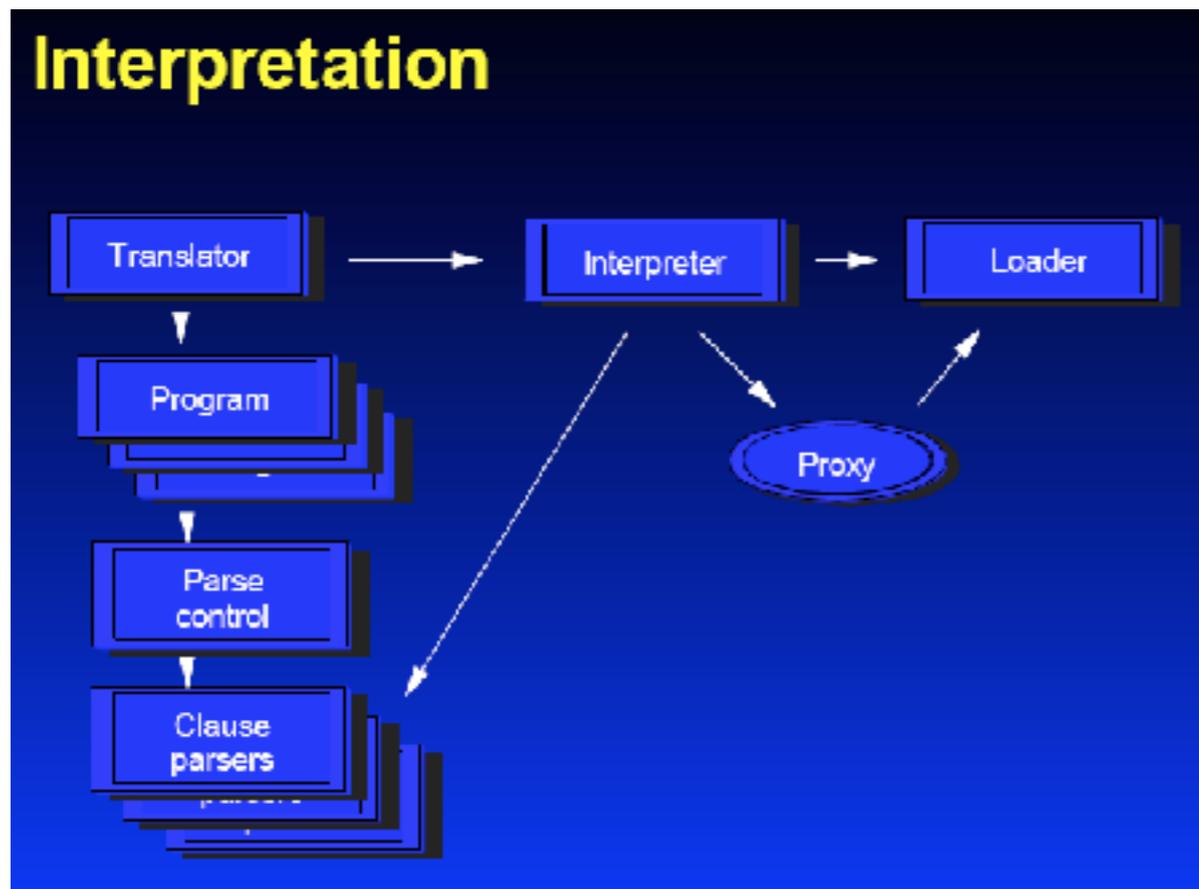
method getSystemAddress() returns String
    return this.systemAddress

method add() returns String
    transaction = this.session.beginTransaction()
    uri = UniformResourceLocator()
    if this.getSystemAddress() = null then this.setSystemAddress('localhost')
    uri.setPrimaryName("http://"this.getSystemAddress():8080/invoker/JNDIFactory
    ")
    this.session.save(uri)
    sip = SystemImplementationService()
    if this.getPrimaryName= null then this.setPrimaryName('dummy name')
    sip.setPrimaryName(this.getPrimaryName())
    sip.setUniformResourceLocator(uri)
    this.session.save(sip)
    transaction.commit()
    say CRDutil.getUser() 'added a SystemImplementationService at' this.getSystemA
    ddress()
    return "success"

```

# Interpreter

- All statement translator classes have an **interpret** method.



© SUN

In the interpreted mode, for each class a proxy ('stub') is created, that contains method bodies that just return, and the properties like in a 'real' class. The proxy is constructed from a byte array;

When the method is called (through reflection) the interpreter executes its body as read from source.

# Tied to Java object model and staticness

```

do
  /* check whether the getter returns an object instance. if it
  * does, we pass it an EditorVisitor instance that handles the
  * editing this of course polymorphically with double dispatch
  * on the indirect object.
  */
  invokeResult = this.globalGetter.getMethod().invoke(this.globalObject, null)
  /* if the result from the Getter invocation is null, we
  * instantiate a new object and also have it accept an
  * editorvisitor.
  */
  if invokeResult = null then
    do
      do
        cls = Class.forName(this.globalGetter.getMethod().getReturnType().getName())
        clz = cls.newInstance()
        (Visited clz).accept(edV)
        edV.getEditor.setFont(this.dialogFont)
        ppp.validate()
        this.panel.validate()
      catch Exception
        say "Exception instantiating object-to-be-edited"
      end
    end
  else
    do
      (Visited invokeResult).accept(edV)
      edV.getEditor.setFont(this.dialogFont)
      ppp.validate()
      this.panel.validate()
    end
  end
end

```

Properties of Involvement Party	
ID	1166
Primary Name	Involvement Party
Abbreviated Name	IP
Description	This can be any party, such as an Ind...
Super Type	Fundamental Object
Published Scheme	Data Concept
Formal Scheme	Data Concept
Rank Within Scheme Rank	110
Class	class.com.ibm.mdm.modelbase...
Classification Source Type	Core Model Classification
Determining Type	Involvement Party
Included In Projects	[VC Advice MS, Sales Activity Manag...
Initial Population Timestamp	1900-01-01 00:00:00
Last Population Timestamp	2005-03-11 15:04:55
Objective Relationship Types	[]
Object Of Relationship Types	[Central Bank CoA Contemporary Co...
Parent In Quoted Tree	Data Concept classFor Fundamental ...
Parent In Schemed Quoted Tree	Data Concept classFor Fundamental ...
Partitioning Relationship Type	

A generic object editor

# Performance

- Without going into microbenchmark discussions, NetRexx is a lot faster than the competition - probably as a result of using plain Java source (so leveraging javac) and a minimal runtime without any proxying of objects, and the 'binary' option, which even leaves much of the Rexx runtime untouched if Java primitive types can be used
- The interpreter is a bit slower, but not much so - and we win that back in development cycle turnaround.
- Arguably, missing dynamic language features like open classes is a pain, specially in regard of the full support of this in Open Object Rexx

# NetRexx is completely written in NetRexx

The language is bootstrapped (starting from Classic Rexx)

A working compiler is needed to compile the compiler - save one!

Advantage:

It can be built on every platform where there is a working Java

- Like eComstation and Blue Lion

Structure of the language translator is clear - interpreter like - and readable. Especially if you are into writing NetRexx

# Building mixed source and JavaDoc

```

COMPILE_COMMAND = java -Xms128M -Xmx256M COM.ibm.netrexx.process.NetRexxC

.nrx.class:
    $(COMPILE_COMMAND) $< -comments -sourcedir -time -keep -replace -pipn -format -warnexit0
    mv *.java.keep *.java _

NRX_SRC      := $(wildcard *.nrx)
NRX_OBJS     := $(NRX_SRC:.nrx-.class)
JAVA_SRC     := $(wildcard *.java)
JAVA_OBJS    := $(JAVA_SRC:.java-.class)
DOCPATH      :=
DOCCLASSPATH :=
DOCTITLE     :=
WINDOWTITLE  :=

.SUFFIXES: .nrx .nry .njp .class .skel .xsl .java .pl

#
# target all compiles the netrexx and java code
#
all:: $(NRX_OBJS) $(JAVA_OBJS)

#
# target clean removes compiled products
#
.PHONY: clean
clean:
    rm -f *.class
    rm -f *.crossref
    rm -f *.bak
    find . -name "*.nrx" | awk '{$$2 = $$1 ; sub ( /\.nrx/, ".java", $$1 ) ; print $$1 }' | xargs rm -f

.PHONY: doc
doc:
    mkdir -p $(DOCPATH)
    javadoc -J-Xmx128M -classpath "$(DOCCLASSPATH)" -private -author -version -breakiterator -use -d $(DOCPATH)
    -bottom $(BOTTOM) -header $(HEADER) -windowtitle $(WINDOWTITLE) -doctitle $(DOCTITLE)

```

## Open Sourcing (2007 - 2011)

Followed IBM's open sourcing process - OSSC

Prepare source code for release

Tidy up & Package, build procedure, arrange testing suite

Formal handover to RexxLA - The Rexx Language Association

# Recent Additions

- Compile from memory string (3.03) - 2015
- Build everywhere where there is Java (3.04) - 2016
- new eco compiler in NetRexx 3.05 (2017)
- Annotations (3.06) - currently in beta

# Publications

Mike Cowlshaw, **The NetRexx Language**, Prentice Hall,  
ISBN 0-13-806332-X

Heuchert, Haesbrouck, Furukawa, Wahli, **Creating Java Applications Using NetRexx**, IBM 1997, <http://www.redbooks.ibm.com/abstracts/sg242216.html>

Mike Cowlshaw, **The NetRexx Interpreter**, RexxLA/  
WarpTech 2000, (netrexxi.pdf)

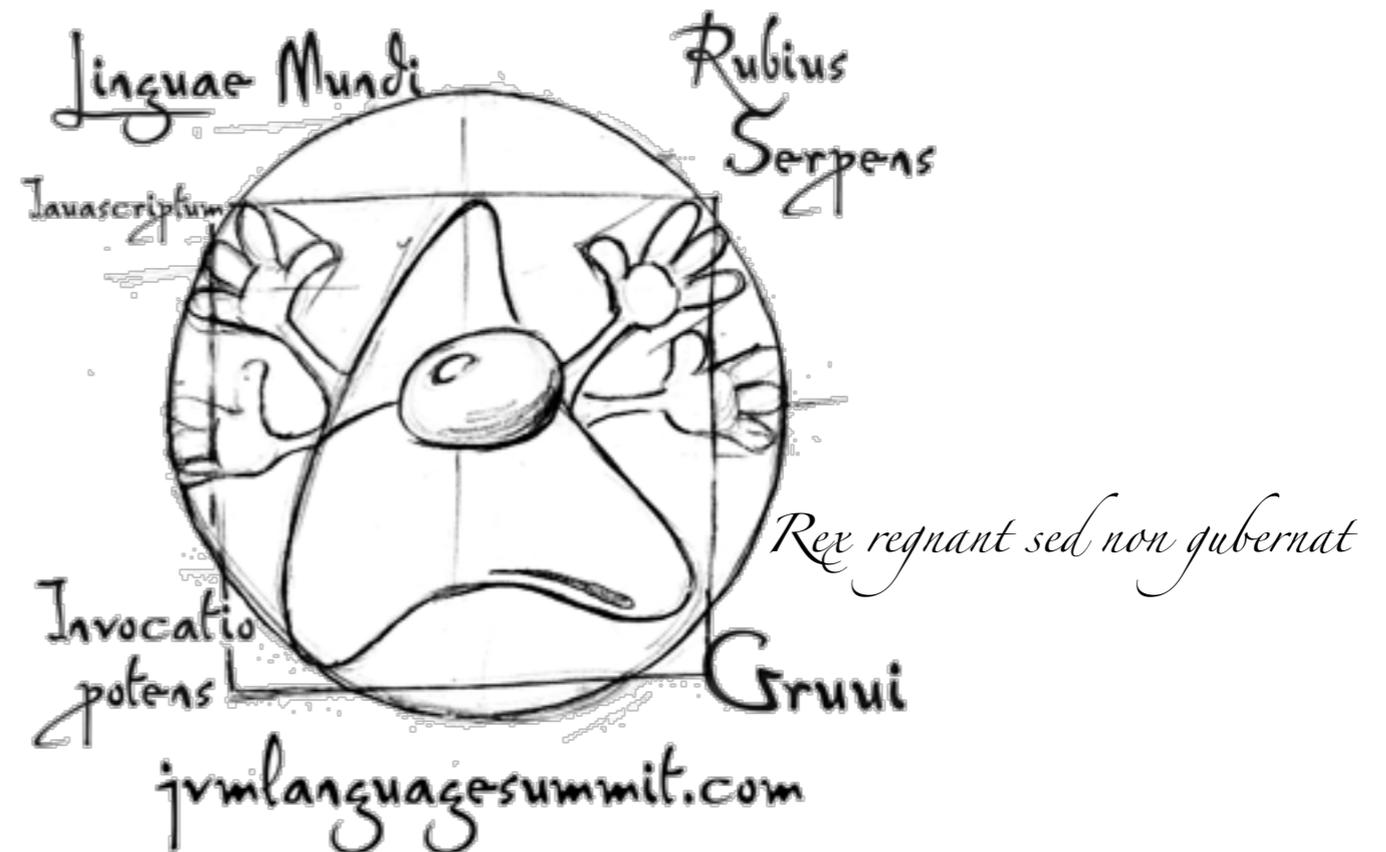


# Contact

[www.netrexx.org](http://www.netrexx.org) tools (emacs & vi modes), documents and other information - watch that space!

[rvjansen@xs4all.nl](mailto:rvjansen@xs4all.nl)

Thanks for your attention!



**“strong typing does not need more typing”**