


IBM COMPILER AND LIBRARY

WALTER PACHL
IBM

IBM Compiler and Library for SAA Rexx/370

Walter Pachi

**IBM VSDL Vienna
Lassallestrasse 1
A-1020 Vienna
 (0043-1-) 211-45-4420
PACHL at VABVM1(.VNET.IBM.COM)**

May 20, 1993

Chapter 1. IBM Compiler and Library for SAA Rexx/370

In a week's time, Release 2 of these products will become generally available. In the following this release's highlights and a number of related matters are described..

1.1 Highlights of Release 2

1.1.1 Support of the Interpret Instruction

When the first Rexx compiler was implemented, it was decided not to support the Interpret instruction was not to be supported. The main reasons were

- the implementation effort involved
- the relatively little use of this instruction

The compiler's Users Guide and Reference did elaborate on ways to circumvent the use of Interpret. For the most frequent use, that is assigning a value to a variable whose name is dynamically determined, a small routine was shipped with the product that could be used for that purpose:

```
varname='ABC'  
Call setvar varname,expression
```

The assembler routine (RXSETVAR) used the variable pool interface to assign the second argument's value to the variable with the name passed as first argument.

This restriction was not removed with the first release of this compiler's successor product because more important user requirements had to be addressed. SETVAR was also provided for the new environment supported by that release: MVS.

However infrequent the use of Interpret may be, there is still the chance (or danger) that a package that you want to compile contains one or more programs that use Interpret and it is not always feasible to modify the programs. And there has been a steady stream of user requirements, asking for the support of Interpret. Therefore, it was finally decided to add this support to the compiler and library.

Incidentally, invocations of setvar can **and should** now be replaced by equivalent calls to the **value** built-in function which has been extended with the capability to set variables (see 1.1.3.3, "VALUE with 2 or 3 arguments" on page 3):

```
varname='ABC'  
Call value varname,expression
```

(rx)setvar is still shipped with this release of the product; the chapter on how to avoid the Interpret has been removed from the User's Guide.

1.1.2 C/370 Library no longer required for compilation

The stated requirements for compiling Rexx clearly indicates that the compiler is written in the C programming language. As the cost of pre-required software must be added to that of the software a customer is interested in, this has probably kept some Rexx users from installing the compiler. Version 2 of the C/370 compiler offers the option to linkedit required library routines with the program that was implemented in C and to ship the "complete" package. After the price adjustments made with release 1 of the current product, exploiting this option is an essential step in making Rexx compilation less expensive.

In addition to reducing cost, compilation of programs became faster. (The library routines have been customized for the specific needs of the compiler.)

1.1.3 Language level 3.48

Mike Cowlshaw's "Red Book" and IBM's SAA Procedures Language Level 2 define what is called language level 4.00 of REXX. (Parse Version returns the language level as the second token.) Level 3.48 is all of 4.00 with the exception of the Rexx input/output functions. These functions have first been implemented on OS/2 and are just about to be provided on VM and OS/400. The language elements that were added for language level 3.48 are discussed in the following.

1.1.3.1 Binary strings, X2B, B2X

A literal string, immediately followed by the symbol b is interpreted as binary string. The literal string must in this case contain only the characters 0 and 1, optionally separated by blanks in certain positions.

```
x='1111 1001'b      )  
x='F9'x            ) these are all the same (on EBCDIC)  
x='9'              )
```

This language extension leads to a slight incompatibility. Before the introduction of binary strings, `x='abc'b` was the concatenation of a constant with the value of variable b. This expression will now cause an error message from the compiler (or raise the syntax condition when interpreted). The instruction `x='1101'b` will, unfortunately, change its semantics without being noticed. **The lesson learned:** Rexx taught me to avoid the variables I used in high school (x, y, z); now I avoid also a, b, and c.

The new built-in functions, X2B and B2X, support the conversion from hexadecimal strings to binary strings and vice versa

```
X2B('A')    --> '1010'  
B2X('1111') --> 'F'
```

Conversions from character strings to binary can be achieved by a two-step process:

X2B(C2X('9')) --> '11111001'

1.1.3.2 Parsing templates +(v), -(v), =(v)

Variables could always be used for literal patterns in parsing templates. Now they can also be used as relative and absolute positional patterns.

1.1.3.3 VALUE with 2 or 3 arguments

The VALUE built-in function has been extended to allow for assigning a value to a dynamically determined variable. Additionally this function can be used to set the value of an "environment variable." (This is supported under VM beginning with CMS Release 6.) The name of the environment must then be specified as the third argument and the value of the first argument must in this case be a variable name that is valid for that environment.

1.1.3.4 Drop (Ivar), Expose (Ivar)

One other use of Interpret was the following **illegal** Rexx snippet:

```
a: Interpret 'Procedure Expose' v1
```

This is invalid because Procedure must be the first instruction of a subroutine, if it is used. Early CMS implementations of Rexx did not enforce this rule, an error that has since been corrected. The reason for using this construct was mainly to cast the list of variables to be exposed into a variable and to use this variable name instead of the long list. This use is now officially supported by using an indirect variable

```
a: Procedure Expose (v1)
```

For consistency, the other instruction that deals with lists of variable names, Drop, has also been extended in the same way.

1.1.4 DBCS symbols (and comments)

With the new release, pure and mixed DBCS strings can be used as symbols, that is variable names, labels, etc.

At this time the remote possibility of a bug was removed: the occurrence of '*' or '/' as bytes within a DBCS string used in a comment.

1.1.5 Smaller executables

The first compiler offered already significant performance improvements. However, compiled programs were, in general, larger than the source programs; sometimes significantly so. Release 1 of the IBM Compiler for Rexx/370 introduced the CONDENSE compiler option use of which results in significant disk storage and I/O savings. With release 2, another little reduction in the size of compiled programs was achieved. Compiling

REXXDX, the program that implements the CMS compiler invocation dialog, shows the following disk requirements:

kBytes

160	Source program
266	CMS REXX
230	REXX/370 R1
221	REXX/370 R2
73	REXX/370 R2 (with CONDENSE)

1.1.6 Improved Compiler Listing

Several improvements have been made to the listing that is produced by the compiler:

- A summary of messages issued and their severity is printed at the beginning of the listing.

```
1 message(s) reported. Highest severity code was 12 - Severe
or, the better alternative:
Compilation successful
```

The user can immediately check the compilation's success.

- The compiler options used are now printed in alphabetical order of their keywords proper (disregarding the NO prefix, where applicable).

Sample Listing of Compiler Options:

Compiler Options

```
CEXEC      (DAMEN EXEC A1)
NOCOMPILE (S)
CONDENSE
NODLINK
NODUMP
FLAG      (I)
LINECOUNT (55)
OBJECT    (DAMEN TEXT A1)
PRINT     (DAMEN LISTING A1)
NOSAA
NOSLINE
SOURCE
NOTERMIAL
NOTESTHALT
NOXREF
```

- A list of flagged instructions is now printed at the end of the compiler listing, if applicable.

1.1.7 Support of VSE

As of this fall, Rexx will also be supported in the VSE environment. It will be possible to run Rexx programs compiled under CMS or MVS in that environment. The support for compiled Rexx will be integrated with the Rexx Interpreter on VSE.

1.2 Performance

1.2.1 Language Features

The speedup for a particular Rexx program depends on the language constructs being used in the program. The following table relates miscellaneous constructs with the performance improvement to be expected.

Programs with a lot of this ... are that much faster than the SPI	
=====	
Arithmetic operations of default precision	9.7

Constants and Variables	5.8

Ref. to built-in functions and procedures	4.9

Changes to variables' values	8.7

Assignments	25.2

Re-use of compound variables	4.4

Host commands	1.0

1.2.2 A Benchmark Program

A program that demonstrates the performance improvements is the following program that computes the number of ways you can place eight queens on a chessboard so that none interferes with the others.

```
/* Position n queens on a chess-board of n*n fields so that no queen **
** can beat any other on the board *****/
Change Activity:
871211 PA Rexxified from the BASIC algorithm supplied by Alfred Gschwend
881104 PA give return code 0 if 92 solutions were found
910919 KH remove test code
*****/
```

```

/*****
00  REM  *** Das Acht-Damen-Problem ***
10  I = I+1
20  D(I) = 1
30  FOR J = 1 TO I-1
40  IF D(I) = D(J) { ABS( D(I)-D(J) ) = I-J THEN 90
50  NEXT J
60  IF I<8 THEN 10
70  R = D(1)*1E7 + D(2)*1E6 + D(3)*1E5 + D(4)*1E4 + D(5)*1E3
80  PRINT ' ->'; R + D(6)*100 + D(7)*10 + D(8);
90  D(I) = D(I) + 1
100 IF D(I) <= 8 THEN 30
110 I = I-1
120 IF I>0 THEN 90
130 END
*****/

Parse version v
Say v

Call time 'R'
cs=cputime()
nq=8
If arg(1)<>' Then nq=arg(1)
n=0
x=''
i=1
sym='0123456789ABCDEFGHIJKLMNO'
ende=0
d.=0
d.l=1
Do nn=1 By 1 While ende<>1
/*call out*/
further=0
Do j=1 To i-1
If d.i=d.j {
abs(d.i-d.j)=i-j Then Leave
End
If j=i Then Do
If i=nq Then Do
n=n+1
/*call out*/
End
Else Do
i=i+1
d.i=1
further=1
End
End
If further=0 Then Do
Do i=i By -1 while(d.i=nq)
End
If i<1 Then ende=1
d.i=d.i+1
End
End
Say x
Say n 'solutions computed'

Say 'DAMEREXX: elapsed: ' time('E') 'cpu: ' cputime()-cs
Exit n<>92

```

The System Product Interpreter needs about 13 seconds to run that program (on a 9121-400). My PS/2 model 95 takes 75 seconds. The following table shows the timing of the same program with the possible combinations of compilers and run time libraries.

	COMPILE Time	EXECUTION Time		
		CMS/REXX	REXX/370 R1	REXX/370 R2
CMS REXX	0.28	1.14	1.16	1.17
	0.29	1.12	1.15	1.15
		1.13	1.16	1.15
REXX/370 R1	0.24		1.09	1.08
	0.24		1.08	1.08
	0.24		1.09	1.07
REXX/370 R2	0.21			1.11
	0.21			1.10
	0.21			1.11

1.2.3 Compiler Options

Some compiler options affect the runtime performance of the compiled programs. These are discussed in the following.

1.2.3.1 CONDENSE

The CONDENSE option causes the compiled program to be stored in a condensed format. This has the following advantages:

1. The compiled program uses less disk space.
2. Preloaded compiled program use less virtual storage.
3. Loading the program requires less I/O activity.
4. Literals in a program become illegible (and un-"ZAP"-able).

On the other hand there are a number of little disadvantages:

- There is a minimum overhead for unpacking the program at execution time.
- The virtual storage required while the program is being executed is larger.
- It takes some time to do the packing at compile time
- The CONDENSE option is mutually exclusive with the DLINK option.

1.2.3.2 TESTHALT

Compiling with the TESTHALT option causes tests to be included in the executable code that determine whether the user has attempted to interrupt the program's execution (by entering the immediate command HI, for example, under CMS). The cost of these tests at execution time is negligible.

1.2.3.3 DLINK

This option results in the most spectacular performance improvement if a large number of external function and subroutine calls are made during a program's execution. Using this option, a program and its external subroutines can be packaged into a module that uses branch-and-link instructions to invoke external subroutines. Avoiding the CMS (or MVS) search order for external routines is the reason for the dramatic performance improvement. A fringe benefit of using this technique is that changes in the program's environment (name clashes with invoked external routines) do not have any effect on the packaged program.

1.3 Testing the Rexx Compiler

Beginning with the first Rexx compiler, the CMS Rexx Compiler, a test project was set up to develop a suite of function test cases to test the language implementation as extensively as possible. Rexx was used to implement a highly automated test environment and to minimize the effort of test case writing.

1.3.1 Original Test Ideas

As any other test, the test cases for Rexx must compare the results from a language construct with the expected results. Results include

- the values of variables after executing the construct to be tested
- flow of control
- error messages
 - at compile time (for errors that are detected by the compiler)
 - at run time
- the contents and layout of compiler listings
- compiler and runtime performance
- etc. etc.

The test project was given significant lead time and could use the existing implementation, the System Product Interpreter, for testing the test cases and for constructing the test environment.

An ideal test case would consist simply of the construct to be tested, for example:

```
(5.6+1.00000000000002)
```

The expected result was either that produced by the Interpreter or that from a "pseudo-implementation" of Rexx (very much like the approach now being taken by the Rexx standardization committee).

Most of the test cases have been constructed to be self-checking. Techniques were developed to automatically handle error situations like Syntax and Novalue conditions being raised. The test environment performs the bookkeeping of successful test runs and the notification about failing test cases. With the completed test suite, human involvement is only required

- to request the execution of the test suite for a particular implementation
- to run those test cases for which human action or attention is required
- to verify, on a glance, on the morning after that no errors occurred
- or to report errors to the developers
- to extend the test suite when an error is discovered or reported or when a new test idea crosses the mind.
- and, of course, to rework the test cases for new implementations or new environments.

1.3.2 Reuse of Test Cases

The test suite has been kept alive over the past years and was extended to test all releases of the compiler in all supported environments (currently CMS and MVS, with VSE to come soon) and other Rexx implementations (such as the interpreters on most IBM platforms). This approach has not only resulted in a very high quality of the compiler products but has weeded out some errors in the other implementations.

A significant effort is, however, involved in keeping the test suite up to date for all implementations it is used for and one has to take care that the number of “generation directives” does not become excessive. Variations to be catered for include

- changes of the language

```
x='123'b /* changes meaning in 3.38 */
```

- implementation improvements

```
x='a'  
l: x=x+1 /* is now a compiler detected error if l is not used */
```

- character set

```
x='F1'x /* is two things on ASCII and EBCDIC */
```

- extensions of the language

```
x=value('x',123) /* new second (and third) parameter */
```

The forthcoming Rexx standard will, of course, be considered for further extensions and customization of the test suite.

1.3.3 Sachertorte

As it is typical for this product, the current release was "finished" quite some weeks before the committed end date. This situation was (again) exploited to our customers' advantage by exposing the compiler to a large number of IBM internal users. This time the development team had to motivate their users to try hard in finding problems, that is bugs, in this very well tested product. A contest was put in place that every person finding one or more reasonably severe errors was to be awarded with a Sachertorte¹. The person who found the most problems is to collect the cake in Vienna where he can meet the developers and testers and can enjoy a few days in not too bad a town. Rewarding customers for problems they find is a process yet to be explored and defined; for the time being we try to deprive them the "pleasure" to find problems.

Meanwhile here is the recipe for the Sachertorte that my wife is using:

Sachertorte allegedly the original recipe from Sacher.
----- translated By Walter Pacht 900531
 (with BJ's help - on the English side).

Ingredients

140 g butter
160 g sugar
180 g ground chocolate
8 eggs
30 g powdered sugar
140 g wheat flour
1 level tea spoon baking powder
200 g apricot jam
200 g icing

Stir butter and sugar to get a foamy cream.

Melt the ground chocolate OVER (not in) hot water, stir well until cooled down

Add the chocolate to the butter/sugar mixture.

Keep stirring until the mixture is thickly foamy

Slowly, by and by add egg yolks, beat heavily until you have a chocolate cream.

Beat whites of the eggs and powdered sugar until stiff and put this on top of chocolate cream.

Mix flour and baking powder, add on top of all the above.

Mix it all cautiously (slowly, carefully).

Fill the dough into a cylindric cake-form that you have coated (on the inner side :- with aluminum foil or baking paper (ours is about 12 inch in diameter).

Bake (use a knitting needle to check if done - it'll come out dry then)

Let the cake cool down, take it out of the form. Heat the apricot jam and smear it on top and on the side and let it soak a little into the cake.

Heat the icing in hot water and cover the cake with it.

Note: Almonds, nuts, cream are NOT to be used in Sachertorte.
Whipped cream is recommended with it.

Bon appetit.

¹ A famous chocolate cake produced (not only) by Hotel Sacher in Vienna.