Compact Rexx - 1999

A recapitulation of CRX 1998

At last year's symposium I described how I was in the fortunate position of deciding for myself what I wanted to program, and that for nostalgic reasons I was coding an implementation in Assembler Language for the DOS operating system. My presentation then can be found on the REXXLA website, but I will summarize now what is needed as introduction to new material.

There is a well-established benchmark program, called REXXCPS, for assessing the execution speed of implementations.

{Display of REXXCPS program. Note independence from operating system performance.}

Last year I said that my ambition to run REXXCPS at a million clauses per second was going to have to wait for new hardware because my 200MHz machine was not getting much more than half that speed. Well, this laptop is a 300MHz machine:

{Display of execution at 1Meg+ clauses per second.}

I will show this compared with commercial products, because you will want to know, but bear in mind that REXXCPS is a test of a narrow aspect of a product; there are weighty issues of reliability, utilities, documentation, debug environment, and so on that effect commercial products.

{Display of execution on Warp with Classic Rexx}

{Display of execution with Personal Rexx 3.0}

Here is a bullet list from last year of the features that make this speed practical, with some comment on each:

- Names compactly numbered so that variable values are in an array with eight byte elements. These numbers are established, based on the different names in the program, before execution of the program begins. In execution it is not necessary to use the spelling of the variable's name in order to access its value.

- Pseudo-code implying a stack, with many tests compiled out. The 127 different values that are the odd numbers that go in a single byte provide a reasonable domain for the different operators needed when executing a Rexx program. Two-byte fields for operands allow for several thousand different variable names and constants in a program. So most of a program can be represented by these one and two byte fields. The order in which they are interspersed determines which operators apply to which operands, so that costs nothing. (Some other formats are needed as well, for instance IF statements give rise to jumps, which have two byte addresses giving the target of the jump.)

- Data representations that dynamically adopt one of several eight byte representations. Eight bytes is a good size for a Rexx value - it can hold all numbers for most programs and the majority of strings. Longer values can be pointed to from the eight bytes.

- Garbage collection with defragmentation and a never-overlay assignment policy. This means that an assignment like A=B never moves a lot of bytes, even when B is long, because it is the eight byte pointer to B which gets assigned to A.

- Alternative interpreter loops working on the same pseudo-code. The TRACE facility, for example, slows execution, and there are other speed inhibitors. Using different parts of the implementation at

different times, according to what actually happens in execution, ensures that the costs to speed are only incurred while the costly features are being used.

- Data type analysis leading to different pseudo-code for different cases, for example uses of the built-in functions with or without checks on the arguments to them.

Most of that list is about good design of the mechanics of an interpreter, rather than about language or compiler features. The last few percentage points in speed come from analysing any individual program before execution to discover decisions that can be made before execution starts, rather than during execution. The program analysis that I described last year concentrated on the built-in functions. In a case like SUBSTR(X,6,1) it is clear that checking the arguments isn't necessary - Rexx does not place constraints on the first argument and the others are obviously numeric and in the right range. SUBSTR(X,J,K) would be a different story.

Since then, I have implemented another piece of analysis - types of comparison. As you know, a comparison A>B can be a numeric comparison or character comparison; the former only if both A and B are numbers. It follows that A>"q" cannot be a numeric comparison because "q" is not a number. Making this decision before execution avoids making it repeatedly during execution.

There are more of these little local analysis decisions that can be made early but in general Rexx does not allow a more global analysis. For example, a variable ABC might seem to have numeric values throughout a program, because all the values assigned to it are numeric constants or the results of numeric operations. However, if there is an INTERPRET instruction in the program, that might put some non-numeric value into ABC in a way that is not discernible before execution of the program.

Global analysis is only possible if we restrict what the programmer can write - and we know that when the MVS Rexx compiler did not allow the INTERPRET statement that was an unpopular feature.

So, if there is not a lot more that can be done, how good is a megaclause per second with REXXCPS on a 300MHz engine? We have seen that it is good in terms of Rexx expectations but that does not tell us if it is good in terms of what the hardware is capable of.

We are fortunate to have REXXCPS as a benchmark but it is only one program and because it heavily uses PARSE and the "typeless" nature of Rexx, it would be difficult to translate it into a program in "C". If we want to compare with "C", and lacking an accepted performance measurement suite of programs, we must settle for measuring just a few basic operations; for example a loop of a billion times doing nothing, or making a ten thousand byte string by concatenating a byte at a time. Even these primitive timing comparisons are subject to a number of influences, for example:

- Even a simple loop in "C" requires a control variable - there is no equivalent to the "DO expression" repetitor of Rexx. Therefore, "C" has to do a bit more work in the loop.
- The popular "C" compiler that I use does a lot of optimizations, including unsafe ones, but has no option to say "produce 16 bit code that exploits 32 bit features of the 486 engine".
- Operations on strings in the "C" language are subroutine calls. This leads to uneven results - string operations are faster in Rexx (CRX) than in "C", even though arithmetic operations on small values are twelve times faster in "C".

{ Demo of simple performance measurements, with Rexx and "C" versions. }

Based on such measurements, my views are:

- An achievable target for the computational part of an execution (as opposed to time spent in commands issued from Rexx) is around four times slower than execution of the natural "C" equivalent. This compares with 10 - 50 times slower that we sometimes get nowadays.

- A fully optimizing compiler could not close the gap from "C" much further. The difference between a mild program analysis (as done by Compact Rexx) and the fullest possible analysis would not translate into a big execution speed difference.

- If enough people were willing to pay enough money for a copy of a PC optimizing compiler of Rexx then it would be good to have one developed, almost irrespective of its effectiveness. However, the technical justification for a genuine optimizing compiler making machine code is thin.

Using pseudo-code as part of the Implementation

Changing the topic to how small a Rexx interpreter can be, I described last year how message texts and the tables that parse the source program can be made small. There was also this remark:

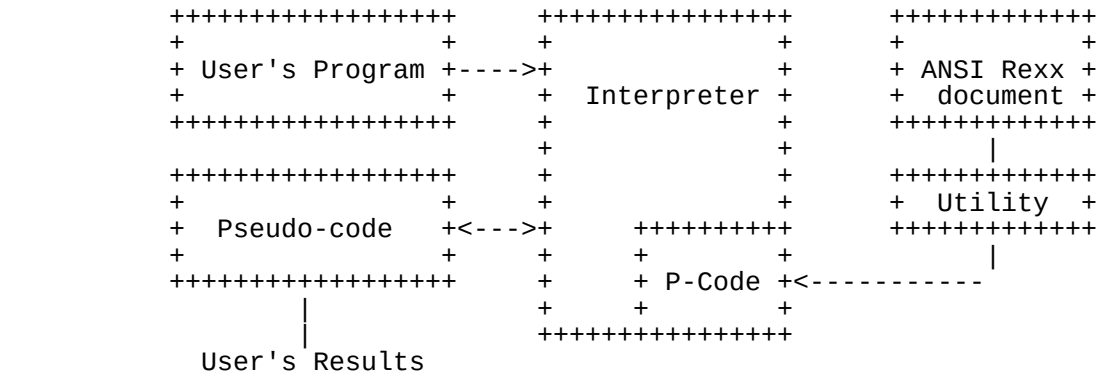
"CRX uses two interpreter loops, essentially to restrict the overheads of tracing to the times when tracing is enabled. There is a case for a third interpreter loop, used when executing Built-in functions. The REXX Standard contains large amounts of coding in REXX, used to define the meanings of Built-in functions and of the ADDRESS instruction etc. So a plausible way of implementing such features in a REXX processor would be to take the code from the Standard, compile it to pseudo-code, and incorporate the pseudo-code in the REXX processor. When the user's program called for a Built-in function to be executed the REXX processor would execute that pseudo-code."

The first new material here is about how that idea worked out.

{Display of the relevant code from the ANSI standard.}

The idea of having part of an implementation as pseudo-code, i.e. some code that is not the machine code of any hardware but is the output of some special compiler or utility program, has a long history. If somebody tells you they have a "portable" implementation of COBOL it may be that they have something written in "C" that can be recompiled but it also may be that the implementation is largely in pseudo-code and the actual bytes don't change when ported across hardwares.

This diagram shows on the left the usual Rexx interpreter picture with source being translated to pseudo-code and that pseudo-code executed. The right of the picture shows an activity that happened earlier when the interpreter was being constructed - source from the ANSI standard was processed to pseudo-code that became part of the interpreter.



There is a choice about how much of the ANSI code to use in this way - for something frequent and simple like SUBSTR it will be better to implement directly in Assembler. However, in general there is not a big speed loss from the embedded pseudo-code approach. The algorithms come down to executing basic operations like comparing and assigning Rexx values - things that would be subroutine calls even if implemented in Assembler. Making the calls from an interpreter loop rather than from plain code does not add much overhead. There is, however, a caveat. The algorithms in the ANSI standard were written to be understood rather than to perform. The code for the WORDS built-in is:

```
$9.3.29: /* WORDS */

   call CheckArgs 'rANY'

   do Count = 0 by 1
     if subword(#Bif_Arg.1, Count + 1) == '' then return Count
     end Count
```

This will repeatedly scan the subject string. For speed, it needs to be re-written; the experienced Rexx programmer would probably use PARSE to split the subject repeatedly into a first word and a new subject.

Fortunately, the algorithms in the Standard are rarely poor for speed in this way.

At first glance, the idea using the Rexx code from the ANSI standard as part of a compact interpreter will fail because of the size of the pseudo-code. There are some 4000 lines of Rexx in the ANSI standard.

How much pseudo-code does a line of Rexx produce? There may be some indication from the Warp Rexx products because they retain the pseudo-code after execution, making it quicker to load the program next time.

{Display of DIR showing Rexx programs with extended attributes.}

Typically, the extended attribute size is around 100 bytes per Rexx line of source. My knowledge of Warp Rexx internals is insufficient to explain this figure, although because Warp has paging of memory the size of pseudo-code is not likely to matter to the user except where it exceeds the 64K limit for extended attributes.

I described the Compact Rexx pseudo-code format last year; one-byte operators, two-byte operand references, reverse Polish ordering. Data on the spelling of the symbols in the original source is also needed. Together this yields about ten bytes per Rexx line.

If we take it that about half of the Rexx from the ANSI standard is to be made into pseudo-code, with the other half representing function better implemented directly in Assembler, then we are talking about 2000 times 10 bytes of pseudo-code. This 20K is still a lot, if we are addressing the "really small interpreter" challenge where all of syntax checking has been done in 4K.

So can the pseudo-code be made smaller?

Rexx with local variables

Here is a little program that I used last year to illustrate the difficulties in analysing the scope of variables:

```
signal on syntax
...
say X
call MySub
exit
MySub:procedure
...
say X
return
syntax: say X
```

The programmer here has deliberately used a procedure to ensure that the variables called X are different variables. But if control reaches the label 'syntax', which of them will X then refer to? The answer is that we cannot decide until execution.

In the case of the code from the Standard, we can do a bit of re-writing if necessary to avoid the features (SIGNAL and INTERPRET) which give rise to scope difficulties. So the utility that turns the code in pseudo-code can analyse which variables are local variables, for example the Count variable in the algorithm for WORDS.

This allows for the top of the Rexx runtime stack to have this content when a routine executes:

```
.....................................................
. Arguments . Local variables . Intermediate results .   Top here
.....................................................
```

A number with a small range, sufficient to cover the total number of arguments, locals, and intermediates, will suffice to identify individual operands when the routine is running. In practice, although the ANSI code as a whole involves several hundred different variables, the worst-case individual routine needs a range of about thirty.

What this means is that we do not need two bytes to reference an operand. The 256 values that a single byte can have are sufficient to cover operand references (for both "assign to" and "load from") , operators, calls to other routines, and a few global variables. I will call this style of pseudo-code "byte-code" to distinguish it. The operands that represent destinations for branching can also almost all be single bytes. (All the branching in a Rexx program is forward, except that SIGNAL, calls , and DO loops can be backward, and a relative branch over 256 bytes of byte-code is a branch over dozens of Rexx statements.)

There are a few complications in byte-code interpretation which I have resolved by making minor restrictions on the way in which the ANSI code is written but I will spare you the details - having the ANSI code as byte-code in the Compact Rexx interpreter does work.

In fact, there are a few of the 256 values not used in this part of the scheme, so we must find a good use for those.

Fragments - subroutines created by the compiler

The idea that memory space can be saved by making some common parts of the user program into subroutines, even though the program was not written in that way, has a long history. I wrote something in 1978 that made PL/I executables half the size of those made by more normal compiling; there is a Technical Report available if anyone is interested.

Recognition of subroutines can save space even for short sequences. The gain with Intel hardware sequences starts when two sequences of eight bytes are replaced by two calls to one subroutine. When byte-code is being compressed two sequences of six bytes will show a gain. (The difference is because hardware routine calls are three bytes long). To make the point that these routines can be small, I will call them fragments of code.

Recognising sequences suitable for fragments is quite complicated, for example because branches into or out of the sequence (as opposed to within the sequence) will make the sequence unsuitable. Exploiting all the potential for changing the source to make extra sequences that match, for example by re-ordering some statements and re-numbering the arguments and re-numbering the variables, and using fragments with their own parameters, is fiendishly complicated.

In the recognition of sequences in the byte-code from the ANSI source, I implemented only a couple of the potential re-organizations. The first one is that if in the original source an argument to a routine is used only as the initial value of some variable, then this argument and the variable it is assigned to can occupy the same place on the stack, making the assignment unnecessary.

```
.....................................................
. Arguments . Local variables . Intermediate results .   Top here
.....................................................
```

The second trick is to notice that in the arrangement above the local variables are in no particular order. So two sequences that have the same operations applied to totally different variables (of different routines) can be done with a fragment subroutine provided the order in which local variables are mapped is such that the offsets of the relevant variables from the top-of-stack are the same.

Experiments would not be experiments if they always came out as you hoped. Fragment recognition was a bit of a disappointment, reducing the byte-code total size by only about 10%.

An example - The ABBREV built-in function.

To make all this more real, I am going to work through the byte-code implementation of ABBREV. I am not suggesting you follow closely since you are never going to need to work with this internal form but the idea is that by noting how often I say some operation is done in one byte you will get a feel for how close the code is to the ultimate in smallness.

Here is the code as written by the ANSI committee:

```
$9.3.1: /* ABBREV */

    call CheckArgs 'rANY rANY oWHOLE>=0'

    Subject = #Bif_Arg.1
    Subj    = #Bif_Arg.2
    if #Bif_ArgExists.3 then Length = #Bif_Arg.3
                        else Length = length(Subj)
    Cond1 = length(Subject) >= length(Subj)
    Cond2 = length(Subj) >= Length
    Cond3 = substr(Subject, 1, length(Subj)) == Subj
    return Cond1 & Cond2 & Cond3
```

Here is what is fed to the utility that makes byte-code:

```
/* $9.3.1: */ BifABBREV:

    Subject = arg(1)
    Subj    = arg(2)
    if arg(3,'E') then Length_ = arg(3)
                  else Length_ = length(Subj)
    Cond1 = length(Subject) >= length(Subj)
    Cond2 = length(Subj) >= Length_
    Cond3 = substr(Subject, 1, length(Subj)) == Subj
    return Cond1 & Cond2 & Cond3
```

Here is the byte-code that gets assembled into the interpreter:

```
;$9.3.1
 db $WholeGE
 ParmsRec <2,1,1101b>
BifABBREV:
 db 6*8+3,Frag34,SUBJ-1,_Length,_Ge,00h,COND1-Tgt,SUBJ,_Length
 db LENGTH_-1,_Ge,00h,COND2-Tgt,SUBJECT,One,SUBJ-2,_Length,_Bifq
 db 0f8h,SUBJ-1,_Seq,00h,COND3-Tgt,COND1,COND2-1,_And,COND3-1
 db _And,_RetB
```

This is assembler syntax. The semicolon precedes a comment and the colon follows a label. The other elements, preceded by "`db`", comma, and "`ParmsRec`" each occupy a byte.

The first two bytes correspond to the original `CheckArgs` and tell us that ABBREV has two mandatory parameters and one optional parameter, which should be a whole number greater or equal to zero. The next byte gives the amount of space to be reserved on the stack when ABBREV is entered. The `Frag34` is a call to a fragment, that being shared code because the built-in LASTPOS starts in a similar way. `Frag34` does the work up until `length(Subject)` is on the stack. The next bytes put `Subj` on the stack and change that value to its length. The `_Ge` compares the top stack items and the `00h` says to put the result as a Boolean value on the stack. The reference to `COND1` is adjusted by `Tgt` to show it is to be assigned to. The rest is similar except for `_Bifq` which invokes a built-in; the following byte says the built-in is SUBSTR with three arguments.

So is this an implementation of ABBREV in 31 bytes plus 6 for the name "ABBREV" in some table of names? We can say that the marginal cost of ABBREV is about forty bytes but full accounting would make ABBREV carry a share of `CheckArgs` which is used by all seventy built-in functions, half of `Frag34`, and some share of the interpreter. An easier question is "How big is the byte-code in total?" The answer to that is 6357 bytes for the 1895 lines I have chosen to take from the Standard - that is 3.4 bytes of byte-code per line of Rexx source.

In summary:

- Embedding pseudo-code generated from Rexx of the ANSI Standard has the advantages of portability and reliability.
- If that Rexx is slightly rewritten to meet a few constraints, the variables can be held on the runtime stack, allowing a tighter form of the pseudo-code with only a few bytes occupied per line of embedded Rexx.
- This pseudo-code compaction prevents the embedded pseudo-code from being a disproportionate part of the size of the interpreter, although that is not of great practical importance. (It would matter if you were transmitting programs with interpreter attached (in case the recipient didn't have Rexx installed) but that assumes you know the recipient's system and that recipients trust you enough to run the program. NetRexx might fill the need better.)

There is a different area where pseudo-code size matters, the area of external routines. For example, the Rexx in the ANSI Standard describes Rexx Arithmetic, starting from the base that only arithmetic on integers is given. Using this code we can run a file of examples of arithmetic on particular values and check the results.

{Demo, running on Warp.}

The reason this is comparatively slow is that the Rexx from the Standard is run as an external routine and its pseudo-code representation is too large to be held as an "extended attribute". This means that it being regenerated from the Rexx source every time the routine is called.

The committee that the ANSI committee evolved into has suggested a language solution. If the source started with an OPTIONS NORELOAD statement it would mean that the pseudo-code from the initial generation of pseudo-code was to be held over in memory to be reused on subsequent external calls.

With pseudo-code at ten bytes per Rexx line, Compact Rexx with the NORELOAD option ought to be practical even in the restricted memory of the DOS system. Regrettably, I cannot demonstrate this yet, which brings us to the topic of completeness and conformance.

Completeness and conformance.

Compact Rexx has a complete design, and a full basic structure implemented, but there are many gaps in the implementation. When you are coding for fun, completeness can represent a chore and sometimes worse than a chore. Here are a couple of examples that I shall dislike implementing.

What do you think this does?

```
abc = copies("0123456789",200)
def = 1000
say substr(abc,def+2,2)
```

The answer would normally be "12", but it would not be if the code was preceded by

```
numeric digits 3
```

The problem is not with substr, which can always cope with argument values large enough to access the longest of strings, but with the `def+2` addition which will decide that 1000+2 is 1E3.

Nobody has ever shown me a program that benefits from using numeric digits less than the default of nine, but the feature has to be supported.

Is this valid Rexx?

```
b="bb"
do a.b = 1 to 10
if a.b>4 then b="cc"
end a.b
```

The answer is that it will run but has a dependency on the initial content of stem A. What we have is a DO loop that changes to a different control variable part way through the loop! It is hard to imagine anything less sensible but it has to be supported. The ANSI committee could not disallow stemmed variables as control variables because they might be used in a sensible way. The nonsensical uses are hard to detect.

If and when completeness of an implementation is achieved, what do we know about conformance to the ANSI standard? The ANSI policy is that implementers will "self-certify", that is they will run tests to allow them to claim conformance. The tests do not come from ANSI. Some languages may be supported by commercial implementation of a test suite. For Rexx, the best prospect is RexxLA. We have some available sources of test data, such as the examples in the Rexx book and the tests used by Regina. There is willingness to organise these into a suite on the RexxLA web site.
I hope we can make it happen.

Discussion Topics

Finally, a couple of things where your judgement and discussion contributions may be helpful.

What are numeric digits greater than nine used for? There are examples like DATE calculations in microseconds from an historic date, or national debt calculations in cents, which can use a few more digits but is there real usage of long numbers in celestial calculations, cryptography, or elsewhere?

The Intel hardware will allow decimal numbers to be held as characters with one digit per character, or with two decimal digits per byte, or as binary conversions with or without help from the floating point engine. The Compact Rexx design for long numbers operates on one digit per character but I have a suspicion that binary is the best for performance. Each hardware operation would then be dealing with nine decimal digits as opposed to one. This probably outweighs the fact that shifting numbers to bring their decimal points into alignment is much faster in the character-per-digit format.

Changing topic, here is a contribution which is on the RexxLA forum:

> Mace Moneta <mmoneta@att.com> wrote:
>
> > I really enjoyed the REXXwishes article by Scott Ophof. I agree with
> > the need for a standard version of REXX, but I'd propose two standard
> > versions. A full ANSI REXX, and a "tiny REXX" for use in environments
> > where the full implementation is not appropriate
> > (on-CPU/microcontroller ROMs instead of tiny Basic or Forth, embedded
> > applications, Palm Pilots, etc.). This dual versioning has worked well
> > for other languages, making them usable across broad product lines.

How significant is the interest in Rexx for organizers? Is it an automatic reaction, that any new environment should have Rexx, or is there a special need? I have looked at the Psion range of products because my first laptop was a Psion, bought because I knew Mike Cowlishaw had one. Psion make a range of organizers and the expensive models are systemically similar to PC's; they have many megabytes of memory and are programmed in C++. However, the inexpensive Psion 3 is still marketed with 1 megabyte and an 8086 compatible engine forty times slower than my laptop, so it has a nostalgic feel.

I have the book "Programming Psion Computers" and it is clear that the manufacturer has a big advantage when it comes to supplying programming support - they can put the support in the Read-Only-Memory. A Rexx interpreter would run as a user program, and be restricted to 64K of code and 64K of data. However, the pertinent questions are not such much about making a Rexx interpreter but about how it would be used. What would users actually want to do? Could those things be done by Rexx alone, without extra packages for managing icons, interfacing with the docking system, and so on?

I hope that discussions will make this clearer, either at this Symposium or on the RexxLA site.