CodeAnalyzer is a generic engine for analyzing and formatting REXX code. This is a tool for the REXX programmer who works with large, complex programs. I release the program in the hope others might find it useful and in the hope others might contribute to its development.

Remember, this program is a work in progress. I place the code in public domain in so far as I have any right to do so.

Doug Rickman
MSFC/NASA
Doug.Rickman@msfc.nasa.gov
August 28, 2003

Good luck! Remember the problem with knowing what you are doing is that you have deluded yourself. 2/17/94

Installation:

After checking the Requirements section, assuming all is well place the program CodeAnalyzer.cmd in the directory of your choice. How to use it is covered in the section "Operation".

Requirements:

This version of the code has been checked out under OS/2 with the IBM Object REXX interpreter. It makes calls only to the built in functions and to the REXXUTIL library provided by IBM. Patrick McPhee has made a public library which provides the same calls and available for other operating systems. The library is called REGUTIL. Mark Hessling has reported that CodeAnalyzer works under Regina using REGUTIL (and runs several times faster than under Object REXX). To use REGUTIL instead of REXXUTIL find the line

```
     DLL = 'RexxUtil' ; LoadFunc = 'sysloadfuncs'
```

at the start of the program and replace "RexxUtil" with "RegUtil". Please note your operating system's sensitivity to paths and naming conventions, and your configuration may force a change in this simplest case instruction.

You must have a REXX interpreter which provides the COUNTSTR( ) function. The "Classic" REXX under OS/2 does not have this function. The newer interpreter available under Object REXX does have this function. OS/2 users may switch between the Classic and Object REXX versions by using BootDrive:\os2\switchrc.cmd. A reboot is necessary for the replacement DLLs to be loaded. One can also use other DLLs that provide this function, such as the RxSCount function in RXU. Those that do not have the COUNTSTR function can easily replace this call with a subroutine of their creation. It is left to the user who does not have the function to provide the substitute subroutine.

You must also have the function SysStemSort( ), which is provided by either REXXUTIL or REGUTIL. If you do not have this you have several options.

Get a newer Rexx DLL containing the function."

Obtain a copy of Patrick McPhee's REGUTIL. Be sure to load the DLL.

Obtain a copy of REXXLIB, which is available for OS/2. Then edit the lines that make the SysStemSort call to use the ARRAYSORT( ) call. I have placed the appropriate arraysort( ) calls in comment blocks within the code. Be sure to load the REXXLIB dll.

Write a subroutine to do the sorting. There are several generic sort say routines available on the web.

Files:

CodeAnalyzer.cmd - program

CodeAnalyzer.ico - pretty icon

CodeAnalyzer.cmd.AnalOCode.txt - Example output from CodeAnalyzer when run on itself.

Documentation.html - This document in .html format.

Documentation.lwp - This document.

Documentation.pdf - This document in .pdf format. Prettier than the .html.

NormalRun.log - Example of output to screen when running CodeAnalyzer.

Acknowledgements:

CodeAnalyzer now incorporates suggestions and/or code from the following:

Peter Skye

Mark Hessling

I am very grateful for the input.

Bug Reports:

Well, first of all I did not release this code so that I could have more work. But I will look at problems on a time available basis. If you do have a problem and want me to look at it run the program with redirection of STDOUT to a file, i.e.

```
CodeAnalyzer YourProgram > errorlog.txt
```

Do this because the output of GenericError( ) will usually exceed the line limit of a simple shell. Therefore a screen capture looses a lot.   Obviously you will also need to send a copy of the offending code.

Using CodeAnalyzer:

Capabilities

Discriminates between internal (labeled) subroutines, built-in functions, library (DLL) functions and other sources.
Recognizes CALL and function usages.
Recognizes a CALL of the form CALL (variable). It does not attempt to determine what values "variable" might take.
Locates where condition trapping SIGNALs are turned off or on.
Warns if the program is recursive, i.e. it references itself.
Warns if an internal subroutine is never explicitly referenced.
Warns if two or more labels are identical.
Maps which subroutines are called by which routines. This is done using two tables. One shows all routines called by each subroutine. The other shows which routines called each subroutine.
Knows the character range of argument lists for CALL and function references.
Labels are retained in case sensitive mode and references to labels are checked in a case sensitive manner.
Recognizes labels which are literals.
Removes line continuations
Handles comments on same line following continuations
Converts multiline comments into a series of commented individual lines.
Knows where all comments are (line and character start and stops)
Handles nested comments
Handles "nested" literals, for example 'a "a" a'
Knows where all literal strings are (line and character start and stops)

The following function libraries are recognized all or in part:
built in (BI) - the standard functions provided by most interpreters
REXXUTIL
REXXLIB (OS/2)
RXU (OS/2)
RXFTP
RXSOCK
VPREXX (VisProREXX)
RXGDUTIL (A GIF graphics library)

Known Limitations:

ObjectREXX constructs, other than to recognize a directive, are ignored.
A comment between the function_name and the opening parenthesis.
INTERPRET instructions are not "interpreted".
Assumes the code being analyzed is working. See the note following step 1 in the Algorithm section of this document.
Not designed to catch coding or logic errors, though it can help in doing so.

Does not handle statements of the form "CALL FUNCTION , ARG" [The first argument is null] properly.

Currently the program does not produce cleanly formatted code as a separate product. Reformatted lines are created and used internal to the program, but are not written to disk in this version of the program. Please see the section Extending the Program for more information if this topic is of interest to you.

Operation:

To execute from the command line

```
CodeAnalyzer.cmd PROGRAM
```

where "PROGRAM" is the REXX code to be analyzed. Of course this assumes CodeAnalyzer.cmd is either in the current directory or along the path. **CodeAnalyzer** will post progress information to the screen and create a file with the extension ".AnalOCode.txt" in the directory of PROGRAM. The distribution archive provides examples of both outputs. The information to the screen is provided in the file NormalRun.log of the. An example of the ".AnalOCode.txt" output is provided in the file CodeAnalyzer.AnalOCode.txt. Both are from a run where "PROGRAM" was CodeAnalyzer.cmd. For a discussion on the output file's contents see the section "Output".

Output:

One might hope that most of the output is fairly self evident. Please note that there is much more output possible than what the delivered code produces. To access the other outputs you will need to look into the code. For example, in the subroutine "Main" you can dump the reformated lines.

Function Reference Table

The table of function references does need a bit of explanation. I will use the following fragment for this discussion.

```
All Recognized Function and Subroutine References
Line Ref_Type  Beg End NAME            SOURCE          BegA EndA |
00034 FUNCTION   4  14 RXFUNCQUERY     "BI Library"      15   33 |
00035     CALL   6  14 RXFUNCADD       "BI Library"      16   62 |
00036     CALL   6  20 REXXLIBREGISTER "REXXLIB Library" 22   22 |
00038 FUNCTION   4  14 RXFUNCQUERY     "BI Library"      15   30 |
```

The "line" column gives the line number in the raw source code.

The "Ref_Type" denotes the nature of the reference used. A *FUNCTION* was done using the syntax

```
rc=function( )
```

A *CALL* was done using the CALL instruction syntax. There are several possible reference types beside these. For example, there is a "Var_CALL", where the CALL instruction has the form

```
CALL (Variable)
```

The "Beg", "End", "BegA", and "EndA" values refer to are the character numbers in the record. These are positions **AFTER** leading spaces have been removed! "Beg" and "End" are the positions where the name of the referenced function. "BegA" and "EndA" are the positions holding the arguments passed to the function.

"Name" gives the name of the function and "Source" gives what library provides this function. "BI Library" means the built-in functions provided by the interpreter. The source for each function is defined in the subroutine LoadKnownFunctions. If you wish to change this list please see the section "LoadKnowFunctions" under "Subroutine Discussions".

If the analyzed source code has more than one reference on a single line this table repeats itself horizontally. Thus it can get rather wide.

Subroutine Maps

The subroutine maps tell what subroutines a given routine calls and which subroutine was called by another routine. This is done using two tables. The following examples are portions taken from an analysis of an earlier version of CodeAnalyzer.

```
            Subroutine Reference Map 1:
         ROUTINE           -          CALLED
_________________________   _________________________
ProgramBegan               - COMMANDLINEEXECUTION
                           - QUOTE
.........................   .........................
COMMANDLINEEXECUTION       - MAIN
                           - QUOTE
.........................   .........................
FINDCALLINSTRUCTIONS       - FINDPARENTHESES
                           - MATCHSUBROUTINE
                           - SIGNALANALYSIS
.........................   .........................
FINDCALLS2SUBROUTINES      - FINDCALLINSTRUCTIONS
                           - FUNCTIONANALYSIS
                           - SIGNALANALYSIS
```

This table shows that the program began and called CommandLineExecution and Quote. The opening code in any program is given the name "ProgramBegan" and this "label" is used until the first label in the program is found. One can then see that CommandLineExecution then called "Main" and "Quote" again.

To learn what routines are calling a subroutine see the second map table. You can see "FindCallInstructions" was called only by "FindCalls2Subroutines" which was in turn only called by "Main". Looking back at the first reference map table we can see that "FindCallInstructions" calls three other routines.

```
            Subroutine Reference Map 2:
         ROUTINE           -       WAS CALLED BY
_________________________   _________________________
CLEARCOMMENTBLOCK          - MAPNMASKCOMMENTSNLITERALS
.........................   .........................
CLEARLITERALSTRINGS        - MAPNMASKCOMMENTSNLITERALS
```

```
.........................   .........................
COMMANDLINEEXECUTION      - ProgramBegan
.........................   .........................
FINDCALLINSTRUCTIONS      - FINDCALLS2SUBROUTINES
.........................   .........................
FINDCALLS2SUBROUTINES     - MAIN
.........................   .........................
FINDDIRECTIVES            - FINDLABELSNDIRECTIVES
.........................   .........................
FINDLABELS                - FINDLABELSNDIRECTIVES
.........................   .........................
FINDLABELSNDIRECTIVES     - MAIN
.........................   .........................
FINDLITERALS              - MAPNMASKCOMMENTSNLITERALS
.........................   .........................
FINDPARENTHESES           - FINDCALLINSTRUCTIONS
                          - FUNCTIONANALYSIS
.........................   .........................
```

Algorithm:

The principal actions of the program are initiated in the subroutine MAIN. The following lists the steps and, gives the call involved and some commentary.

1 Read the raw source file into memory [rc = ReadRawSource(in)]

In concept there appears to be an ambiguity in REXX interpreters about nesting comments and quotes inside of each other. I have chosen to assume that bounding comments are to be found first.

Recognizing comments and quotes is the single problem restricting **CodeAnalyzer** to working code. I currently assume comments and quotes are balanced. In code that does not work, this is not necessarily true. To recognize and handle unbalanced situations would require changes in MapNMaskCommentsNLiterals( ). Other than this, there is no real necessity for the code being analyzed to already work.

2 Find the comments in raw source. [rc = MapNMaskCommentsNLiterals('RAW_C',)]

3 Find the literal strings in the raw source.' [rc = MapNMaskCommentsNLiterals('RAW_L',)]

It is now possible to reformat the raw code into a consistent pattern.

4 Make the logical lines.' [rc = MakeLogicalLines( )]

5 Find the comments in the logical lines.
[rc=MapNMaskCommentsNLiterals('LOGICAL_C','MAP')]

6 Find the literal strings in the logical lines.
[rc=MapNMaskCommentsNLiterals('LOGICAL_L','MAP')]. There is a copy of the ith new, clean line in the variable LogicalLineI.i LogicalLine1.i holds that logical line with comments removed.LogicalLine2.i holds that logical line with comments and quotes removed. SourceIndex.i is the line number in the raw source for the ith logical line. The

position and contents of the comments and literal strings for the line are in the compound variables "Commnet." and "Literals.".

7 Labels and directives are then found. [rc = FindLabelsNDirectives( )]

8 The list of known functions, i.e. the DLL libraries, is loaded. [rc = LoadKnownFunctions( )]

9 The list of default conditions is loaded. [rc = LoadDefaultConditions( )]. This list: ANY, ERROR, FAILURE', HALT, SYNTAX, etc, is not used by the existing code. It is expected to be used in the subroutine SignalAnalysis.

10 Find all references to functions and subroutines. [rc = FindCalls2Subroutines( )]. This is the heart of the logic mapping operation. Information is stored in the "FRef.".

11 Write the contents of the reference tables. [rc = WriteFRefTable( )]

12 Map the relationship between subroutines. [rc = SubroutineAnalyzer(in)]

Rational

Much of the existing code reflects my desire to extend the analytical part of the program. For example, I would like to be able to find all variables used in a specific subroutine and compare that to the map of subroutines and their exposed variable lists. I have also tried to consider the future needs that might arise as the program is extended.

Major Variables:

Many of the variables are compound variables. This will be denoted by either ending in a ".", such as "LogicalLineI.", or by an internal ".", as in "LogicalLineI.". I have placed code in Main( ) that lists the contents of the LogicalLine-, Comment-, and Literal- series of variables. WriteFRefTable( ) shows how to use the FRef- variables.

Created in ReadRawSource( ).

data. - Original source code.

Created in MapNMaskCommentsNLiterals( ).

dataEdited1. - Source after replacing all comments with blanks.

dataEdited2. - Source after blanking comments and literal strings.

Created in MakeLogicalLines( ).

LogicalLineI. = Original source code.

LogicalLine1. = Comments are blanked out.

LogicalLine2. = Comments and literal strings are blanked out.

SourceIndex.j = First line in original source of logical line j.

Comment.i.0 = Number of comments in line i.

Comment.i._Str.k = Character position for start of comment k in line i.

Comment.i._End.k = Character position for end of comment k in line i.

Comment.i._Txt.k = Text of comment k in line i.

Literal.i.0 = Number of literals in line i.

Literal.i._Str.j = Character position for start of literal k in line i.

Literal.i._End.j = Character position for end of literal k in line i.

Literal.i._Typ.j = Type of literal k in line i (S|D - single or double).

Literal.i._Txt.j = Text of literal k in line I.

Notes -

LogicalLines are lines after editing out of continuations, semicolons and blank lines.

Created in FindLabels( ).

Label.i = Line# || Type ("STRING"|"SYMBOL") || FunctionName

Notes - Since most programs may not name the initial routine I have chosen to refer to the intitial routine by the lablel "ProgramBegan".

Created in FunctionAnalysis( ).

FRef.i.0 = Number of functions referenced in line i.

FRef.i._Str.k = char 1 in name of kth function referenced in line i.

FRef.i._End.k = Last char of name of kth function referenced in i.

FRef.i._Txt.k = Text string (name) kth function referenced in line i.

FRef.i._Typ.k = Type of function, kth function referenced in line i.

FRef.i._Open.k = Postion of "(" for kth function referenced in line i.

FRef.i._Close.k = Postion of ")" for kth function referenced in line i.

FRef.i._Knd.k = Nature of reference, subroutine call or function.

Notes -

1. Positions are relative to the first non-blank character in the line.

2. FRef.line._Open.k = FRef.line._Close.k when the reference is done using the CALL instruction and there are no arguments passed.

3. For CALL instructions FRef.i._Open.k and FRef.i._Close.k give positions of first and last characters of argument string. If there are no arguments FRef.i._Close.k = FRef.i._Open.k.

4. For a CALL (variable) FRef.i._Str.k and FRef.i._End.k are the same as FRef.i._Open.k and FRef.i._Close.k.

5. FRef.i._Knd.k = "CALL" | "FUNCTION"

6. For CALL instructions FRef.i._Open.k and FRef.i._Close.k are computed after all comment blocks have been deleted.

Extending the Program

Design considerations

Subroutines are given in alphabetic order.

Each subroutine should be isolated using a PROCEDURE instruction. Be sure to expose the default variable list. In other words be sure to do

```
procedure expose (DefaultExposeList)
```

Be sure to update the SUBROUTINEHISTORY variable upon entering each subroutine.

```
SubRoutineHistory = 'TabulateCommentsNQuotes' SubRoutineHistory
```

Be sure to remove the current subroutine name EVERY TIME the subroutine returns!

```
parse var SubRoutineHistory . SubRoutineHistory
return 1
```

Reformatting

If your interest is in reformatting REXX code look into the subroutine MAIN. By the line "say 'Finished finding literals in logical lines.' all comments, line continuations and quoted strings have been identified and a consistent, though unformatted, line of code created. There is a copy of the new, clean line in the variable LogicalLineI. LogicalLine1. holds the logical line with comments removed. LogicalLine2. holds the logical line with comments and quotes removed. The contents of comments and quotes are available. To see how look at the code following the comment "Debug aid and illustration ..." that immediately follows the above indicated line.

Logically, a line reformat operation would be inserted in this location. Remember, if the reformat operation modifies the line numbers the "Comment." and "Literal." indices will have to be updated.

Desired or Future Developments

There are several things I would like to implement or need to enhance.

The program currently does not handle a comment between the function's name and the opening parenthesis. To do so reference Comment.i._Str. and Comment.i._End. and see if one of the comments fills the space.

Handling unbalanced comment and quote blocks would remove the restriction that **CodeAnalyzer** be run only on working programs. See the note following step 1 in the Algorithm section of this document.

I would like to handle SIGNAL instructions more completely. This is why the list of default "CONDITONS" is provided.

Variable Analysis

I would like to be able to find all variables passed or used within a specific subroutine and compare that to the map of subroutines and their exposed variable lists. Detecting the variables is the real problem.

It might (???) be done using SysDumpVariables( ), which is part of REXXUTIL. How to implement this is a question! SysDumpVariables( ) gives a list of all variables within the current scope and have already been set. Thus

```
   avar = 'b'
   SysDumpVariables('afile')
   bvar = 'b'
```

knows about "avar" but is unaware of "bvar".

Output of SysDumpVariables( ) is either to a file or to standard out. Presumably a named pipe could also be used but there is no standard, OS independent library providing named pipe handling. Assume we will output to a temp file.

First we must find the range of each subroutine in the source being analyzed. Conceptually, do this by looking through the code in from end to beginning looking for a label. For each label found check to see if the line above it is either a RETURN or EXIT instruction. If not this is an alternate entry point into a subroutine and can be ignored. If either RETURN or EXIT is found the lines from the current line number to the label define a subroutine. In practice it is not necessary to actually read through all the data. Just use "Label." The first word is the line number holding the label.

Having defined the limits of a subroutine in the source then it must be "executed" in order for SysDumpVariables( ) to recognize the variables. How to do this without endless syntax errors I have no idea.

An alternative to using SysDumpVariables( ) is to expand the logic used in GenericError( ) in the section where "`if GenericErrorQUIET = 'DECODE' then do` ...". The current logic is rudimentary at best.

Subroutine Discussions

LoadKnownFunctions

The source of "known" functions is the subroutine "LoadKnownFunctions". This consists of two parts. The first is a list of libraries, which is stored in the compound variable "FunctionLib.". To add or eliminate a library edit the "FunctionLib." list, change FunctionLib.0. Any library not in "FunctionLib." will not be used in the analysis of function references. When adding a library BE SURE TO USE UPPER CASE!

The second part of "LoadKnownFunctions" is a series of entries having the form "Function._LibraryName.FunctionName = 1". To add or delete a function within a library, edit the corresponding line. To ignore a function replace the "1" with any other value. The checking of functions versus the known functions is done in the subroutine MatchSubroutine.

GenericError

CodeAnalyzer incorporates the subroutine GenericError. This routine provides the programmer information in the event of failures. In addition to line number of failure, CONDITION: SYNTAX:, INSTRUCTION:, SIGNAL:, DESCRIPTION: and STATUS:, it gives the current value of variables and the subroutine history to the point of failure. I find these last two details very helpful in debugging code.

Operation of the subroutine is controlled by the variable "GenericErrorQuiet". This variable is set on entry to CodeAnalyzer. For a discussion of other options see the comments in the subroutine.

In order for the subroutine history to be available the variable SUBROUTINEHISTORY is created. On entry to each subroutine the name of the subroutine is prepended to the variable. On leaving the leading word of the variable is removed. When PROCEDURE instructions are used the SUBROUTINEHISTORY variable must be exposed.

A simple illustration of a GenericError output follows. The text in red is the information provided by GenericError. The cause of the failure in this case was leaving a VisProREXX call in the program when it was not being run under VisProREXX.

```
Read 3462 lines from source file,
D:\source\VisProSource\CodeAnalyzer\CodeAnalyzer.cmd.
Finished finding comments in source.
Finished finding literals in source.
Finished making 2957 logical lines.
Finished finding comments in logical lines.
Finished finding literals in logical lines.
Finished finding labels and directives.
Finished loading table of known functions.
Finished loading table of default conditions.
Finished finding calls to subroutines.
Finished writing function references table.
Finished with the subroutine analyzer.
Analysis of the code: call _VPAppExit /* This will force an exit
without opening a panel.   */
LIT "_VPAppExit" = _VPAPPEXIT
LIT "This" = THIS
LIT "will" = WILL
LIT "force" = FORCE
LIT "an" = AN
LIT "exit" = EXIT
LIT "without" = WITHOUT
LIT "opening" = OPENING
LIT "a" = A
VAR "panel." = PANEL.
LIT "panel" = PANEL
```

```
 A serious REXX ERROR has occurred!  I do not know what.
 Other information for a programmer's use:
 The line that generated this error is: 227
 " call _VPAppExit /* This will force an exit without opening a panel.
  */ "
 Subroutine history to point of failure, most recent first:
       CommandLineExit()
   Begin_Program()
 Condition: SYNTAX
 Instruction: SIGNAL
 Description:
 Status: OFF
 RC: 43 SYS0043: Drive %1 cannot locate a specific area or track on
the disk.
```

Change History

September 17, 2003

Several minor editorial changes. This will probably be the last version for at least the next several months.

September 8, 2003

Added notes about SysStemSort( ).

Changed September 2, 2003

Changed to SysFileDelete( ) to delete files.
Output file name is now Program.AnalOCode.txt.
Added QUALIFY to list of built in functions.
Expanded explanation of logic and how to change the list of known functions.
Deleted the "_CL" options from MapNMaskCommentsNLiterals( ).
Fix bug that occurred when a comment contains an single quote mark, i.e. an apostrophe.
Output was correct but an inappropriate error message was repeatedly given.
Fix bug that occurred when the first line of program was not a comment.

August 26, 2003

Patch GenericError( ) for "DECODE" analysis of lines containing quote marks.
Modified MapNMaskCommentsNLiterals( ) to handle a single line. This code has not been checked out. It is there for future development.
Modified FindCallInstructions( ) to handle syntax such as "CALL function, arg".
Significant edits to this document.
Numerous minor bug fixes and changes to comments in the code.

August 22, 2003

Patch for lines starting with "/" within a multiline comment.
Recognized the statements of the form "CALL FUNCTION , ARG" are not handled properly. I didn't even know this one was legal!?! It certainly does not match my ORexx documentation.
Modified the loading of DLLs so that user is warned if DLL not found.