The Search Order for External Rexx Files*

Josep Maria Blasco

Espacio Psicoanalítico de Barcelona
Balmes, 32, 2º 1ª – 08007 Barcelona
jose.maria.blasco@gmail.com
+34 93 454 89 78

May the 16th, 2023

Abstract

We start by studying a subtle *anomaly* in the Open Object Rexx (ooRexx) interpreter: a `::Requires "../filename"` directive or a `Call "../filename"` instruction will not work as documented. We point to the places in the interpreter source code where this behaviour is implemented, and consider two opposed possibilities: fixing the anomaly, or deciding that it is a feature, and then fixing the documentation.

To be able to reach an informed decision, we embark on a series of tests: these allow us to compare the behaviour of the `Call` instruction in seven different Rexx interpreters and under three operating systems. We also study the working of the Windows `CMD.EXE` interpreter, of the Windows `SearchPath` API, of two C/C++ compilers, and of Python's `pathlib` module.

After classifying and analysing the results of our tests, we will be in possession of an ample perspective. We will then introduce a set of ooRexx classes that will allow us to simulate the search order algorithms of all of the interpreters and environments we have tested, and to create new search algorithms. In particular, we will present a class that implements an enhanced search algorithm for ooRexx that fixes the anomaly and exhibits some interesting additional properties.

We will finish by demonstrating a proof-of-concept implementation of a pluggable external search algorithm system.

*URL of this document: `https://www.epbcn.com/pdf/josep-maria-blasco/2023-05-16-The-search-order-for-external-rexx-files.pdf`. Presented to the *34th International Rexx Language Symposium*, held in Amsterdam and online from the 14th to the 17th of May, 2023.

Acknowledgements

I would like to start by thanking Joan Batet, Lisa Mc Connell, Silvina Fernández, Xavier Navarro, David Palau and Francesc Rosés, who have contributed to enhance this document by patiently reading several drafts, finding typos, indicating paragraphs that were unclear, and suggesting all kinds of corrections and enhancements.

I would also like to give my thanks to Erich Steinböck, who pointed me to the relevant parts of the interpreter code; to Rony G. Flatscher, who was especially kind and encouraging, and patiently introduced me to the arcanes of preparing and submitting documentation, code and test patches for ooRexx; to René Vincent Jansen, who invited me to participate in the RexxLA Architecture Advisory Council, a.k.a. Architecture Review Board (ARB), and provided resources in the GitHub `rexx-repository` of the Rexx Language Association for me to store my programs, test results, etc.; to the members of the Architecture Review Board itself, for their critical comments and encouragement; to my colleagues at Espacio Psicoanalítico de Barcelona, for bearing with me while I submerged myself in this research, listening to my musings, and being loving and supportive; to the participants of the different mailing lists, especially the ARB list, the developers list, and the RexxLA list; to the Rexx Language Association, for stimulating my creativity; and, finally, to the ooRexx developers, for maintaining and enhancing a wonderful version of the Rexx language

Contents

Acknowledgements **2**

Introduction **5**
References and sources 6

1 An anomaly **6**
1.1 Simple calls 7
1.2 Downwards-relative calls 8
1.3 The anomaly: upwards-relative calls 9

2 A technical explanation **11**
2.1 In the Unix-like side of things 12
2.2 In the Windows side of things 13

3 How to interpret the anomaly **15**
3.1 As an interpreter bug 15
3.2 As a documentation bug 16
3.3 A limitation that is very difficult to fix 16
3.4 An internal inconsistency 16
3.5 Elements for a decision 17

4 A number of tests **17**
4.1 Interpreters and operating systems 17
4.2 Two bugs in two interpreters 18
4.2.1 The SAA bug 18
4.2.2 The `hasExtension` bug 19
4.3 The directory structure 20
4.4 Types of tests 23
4.4.1 Common tests 23
4.4.2 Drive-relative tests 24
4.5 Special tests 26
4.5.1 The Windows Command Line Interpreter (`CMD.EXE`) . 26
4.5.2 The Windows `SearchPath` API 26
4.5.3 Checking the C/C++ compilers 26
4.5.4 Python `pathlib` and backslash-relative filenames . . . 27

5 Classifying and analysing the results **28**
5.1 Eliminating extensions 29
5.2 Equivalence classes 30
5.2.1 Class 1: Regina, REXXSAA (amended), CMD.EXE . 30

5.2.2 Class 2: Object Rexx for OS/2 31
5.2.3 Class 3: ooRexx (5.1.0 beta) and Windows `SearchPath` 31
5.3 In summary 31

6 Modelling external search algorithms 33
6.1 Location-first and qualifier-first algorithms 33
6.2 Location-exception and qualifier-exception clauses 34
6.3 The composition operation 34
6.4 The `ooRexxExternalSearch` class 35
6.5 The `ReginaRexxExternalSearch` class 35
6.6 The `driveRelative` boolean attribute 36
6.7 The `ooRexxEnhancedExternalSearch` class 36

7 A pluggable external search system 36

8 Further work 37

9 Conclusions 38

Appendices: Test results 39
Regina Rexx under OS/2 40
Object Rexx under OS/2 41
Classic Rexx under OS/2, with the SAA bug fixed 42
Classic Rexx (REXXSAA) under OS/2 43
Open Object Rexx under Ubuntu 44
Regina Rexx under Ubuntu 45
Open Object Rexx 5.0.0 under Windows 46
Open Object Rexx 5.1.0 beta after commit r12651 under Windows . 47
Regina Rexx under Windows 48
The Windows Command Line Interface (`CMD.EXE`) 49
The Windows `SearchPath` API 50

Introduction

All the results and findings presented in this article are the consequences of investigating a very simple fact, something apparently unimportant, banal: when we use the Open Object Rexx (ooRexx) interpreter, a `Call` instruction of the form

```
Call "../routine"
```

will not work as expected (and documented): instead of searching first in the same (or caller's) directory, then in the current directory, and finally in a number of paths (including the `PATH` environment variable), the search will be limited to the current directory. The same is true when a `::Requires` directive has a similar form, for example

```
::Requires "../myClass"
```

This *anomaly*[1] of the ooRexx interpreter is highly unexpected, apart from being contrary to the documentation. We will devote the first of our sections, called *An anomaly*, on page 6, to the detailed description of this behaviour.

Section 2, *A technical explanation*, on page 11, examines the technical causes of the anomaly. We will submerge in the entrails of the interpreter code, where we will find a number of unexpected surprises.

In section 3, *How to interpret the anomaly*, on p. 15, we ponder whether to interpret the anomaly as an interpreter bug or as a documentation bug. We also point to a striking internal inconsistency in the behaviour of the ooRexx interpreter. Our investigation will not be conclusive; we will end up by deciding that we need to embark ourselves in a quest for more information.

We will produce this new information by implementing and running a number of tests. These tests, their types and their variations are described in detail in section 4, *A number of tests*, on page 17.

In section 5, *Classifying and analysing the results*, on page 5, we will classify and analyze the results of our tests. This will allow us to obtain a quite ample perspective over the general problem of the external search algorithms.

Section 6, *Modelling external search algorithms*, on page 33, will be devoted to presenting a set of ooRexx classes that model the behaviour of external search algorithms. We will introduce classes for all the algorithms studied in the previous sections, and, additionally, we will introduce a new, enhanced, class that completely fixes the anomaly.

[1]Which also extends to `Call` statements of the form `Call "./Routine"`, or requires directives of the form `Requires "./myClass.rex"`.

In section 7, *A pluggable external search system*, on page 36, we will introduce an experimental system that will allows us to install a search algorithm and then call a routine, knowing that all the `Call` statements present in this routine (and in any subroutine called from this same routine, and so on, recursively) will be resolved using the installed algorithm.

In section 8, Further work, on page 37, we point to possible avenues to widen our investigation.

Finally, in section 9, Conclusions, on page 38, we present the conclusions of our work, after a quick recapitulation of our journey.

The Appendices, on page 39, collect a simplified version of the test results.

References and sources

Instead of building a bibliography, which can be cumbersome to use, we are giving all the required references to manuals, online help files, etc., when the respective documents are quoted.

All the test programs, results files, etc. referenced in this document can be downloaded from `https://www.epbcn.com/pdf/josep-maria-blasco/2023-05-16/` and from `https://github.com/RexxLA/rexx-repository/tree/master/ARB/standards/work-in-progress/search-order`.

The reader might also want to take a look at the accompanying presentation document, a set of slides in the 16:9 format.[2]

1 An anomaly

Our research begins with the study of an *anomaly* in the Open Object Rexx interpreter (ooRexx); this first section is devoted to the presentation and study of that anomaly.

As it is well known, when a `Call` instruction or a `::Requires` directive refer to an external file, there is a *search order algorithm for external files* that stipulates that the "same" (or caller's) directory should be searched first, then the current directory, and then a number of other directories, ended by those specified in the `PATH` environment variable.

All the relevant details are contained in the first paragraphs of the "Locating External Rexx Files" section of the ooRexx manual, which we copy below:

[2]`https://www.epbcn.com/pdf/josep-maria-blasco/2023-05-16/2023-05-16-The-search-order-for-external-rexx-files-Slides.pdf`

7.2.1.1. Locating External Rexx Files

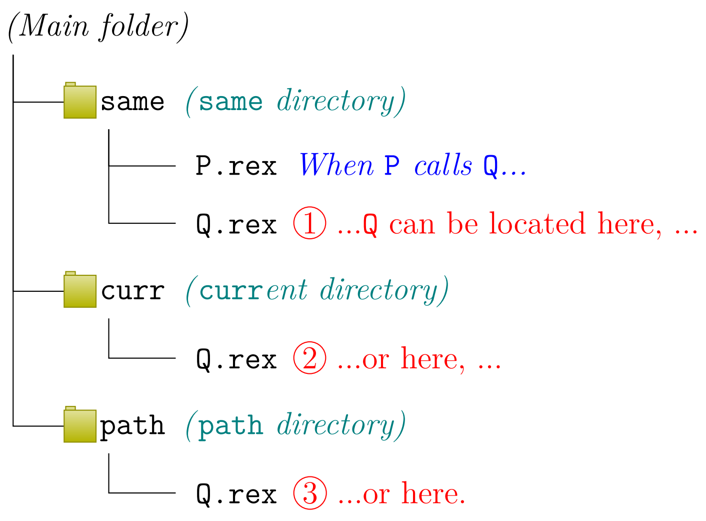
Rexx uses an extensive search procedure for locating program files. The first element of the search procedure is the locations that will be checked for files. The locations, in order of checking, are:

1. The same directory the program invoking the external routine is located. If this is an initial program execution or the calling program was loaded from the macrospace, this location is skipped. Checking in this directory allows related program files to be called without requiring the directory be added to the search path.
2. The current filesystem directory.
3. Some applications using Rexx as a scripting language may define an extension path used to locate called programs. If the Rexx program was invoked directly from the system command line, then no extension path is defined.
4. Any directories specified via the `REXX_PATH` environment variable.
5. Any directories specified via the `PATH` environment variable.

[*Open Object Rexx Reference* (rexxref.pdf), release 5.0.0, section 7.2.1.1]

1.1 Simple calls

To simplify our argumentation, we can assume, in the following, and without loss of generality, (a) that the program was invoked from the command line (and therefore the extension path is absent); (b) that the `REXX_PATH` environment variable is empty, and (c) that the `PATH` environment variable contains only a single directory. We will additionally assume that the same directory is called precisely `same`, that the current directory is called `curr`, and that the only directory in the path is called `path`. If we place these three directories inside a main folder, we will get the following structure.



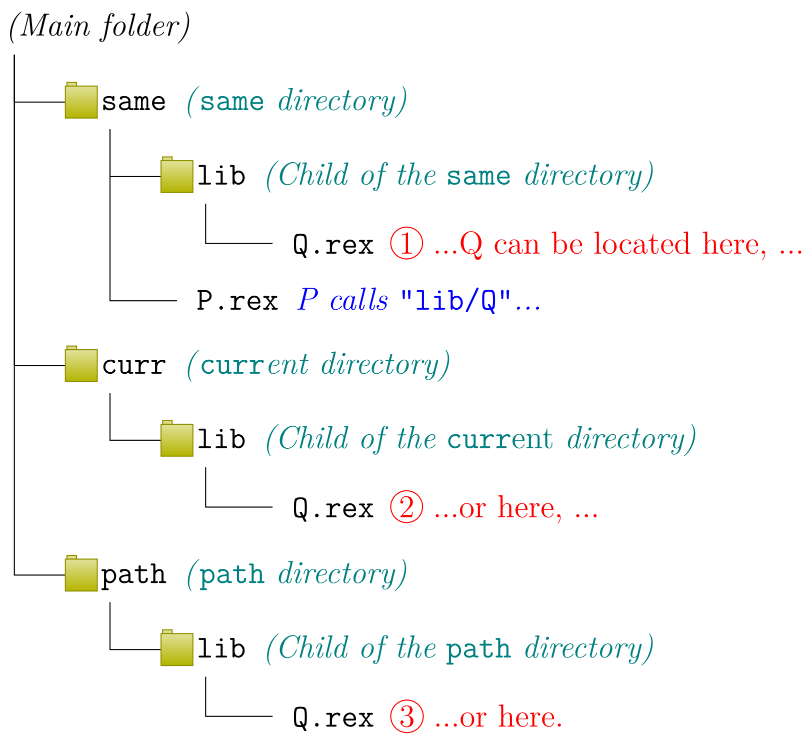
When a program `P` located in the `same` directory issues a `Call Q` instruction, directories `same`, `curr` and `path` will be searched for `Q`, in that order, until `Q` is found. If `Q` is not found, a syntax error (error 43.1: `Could not find routine "Q"`) will be raised.

This set of simplifying assumptions will allow us to focus on the important questions without getting lost in the details (very long paths, etc.), and will come in very handy later, when we will have to prepare a number of tests (p. 17).

1.2 Downwards-relative calls

Now imagine that we have a *library* of programs, collected in a certain *child directory* of either `same`, `curr` or `path`. This subdirectory is called `"lib"`. We could write `Call "lib/Q"` (or `Call "lib\Q"` if under Windows or OS/2), and expect that the search algorithm searches for `Q` first in the `lib` subdirectory of `same`, then in the `lib` subdirectory of `curr`, and finally in the `lib` subdirectory of `path`. In other words, the filename `"lib/Q"` will be *relative* to `same`, `curr` and `path`, in that order: `Q` will be searched for in `same/lib`, `curr/lib` and `path/lib`.

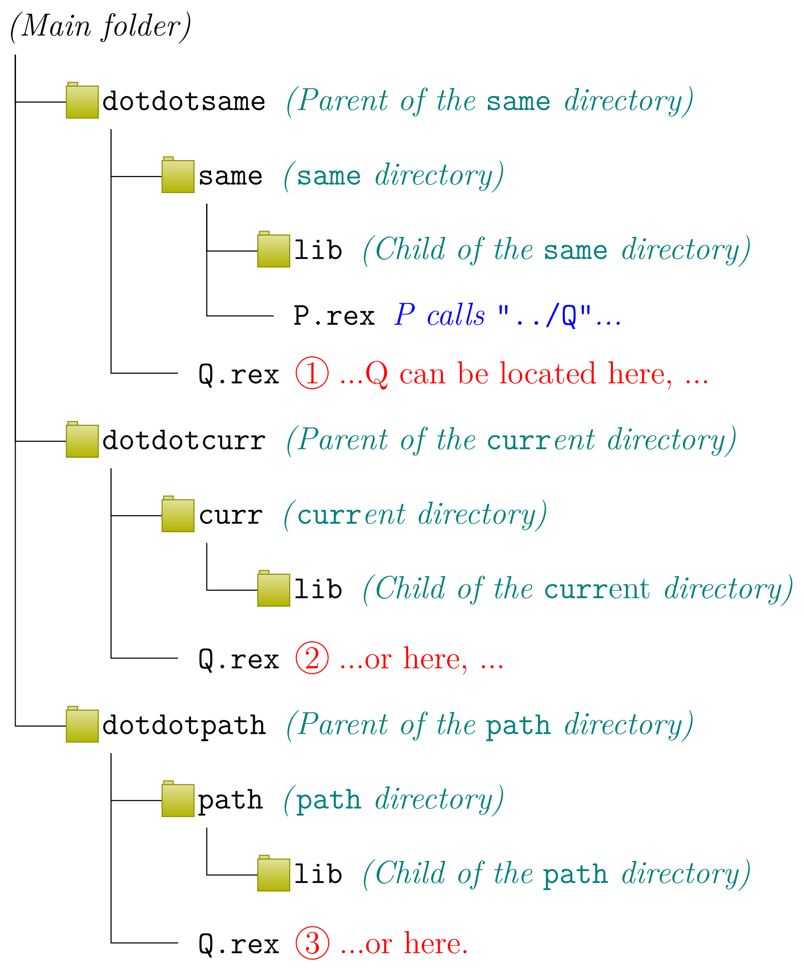
Indeed, this is what really happens. We can visualize the different possible combinations as follows:



1.3 The anomaly: upwards-relative calls

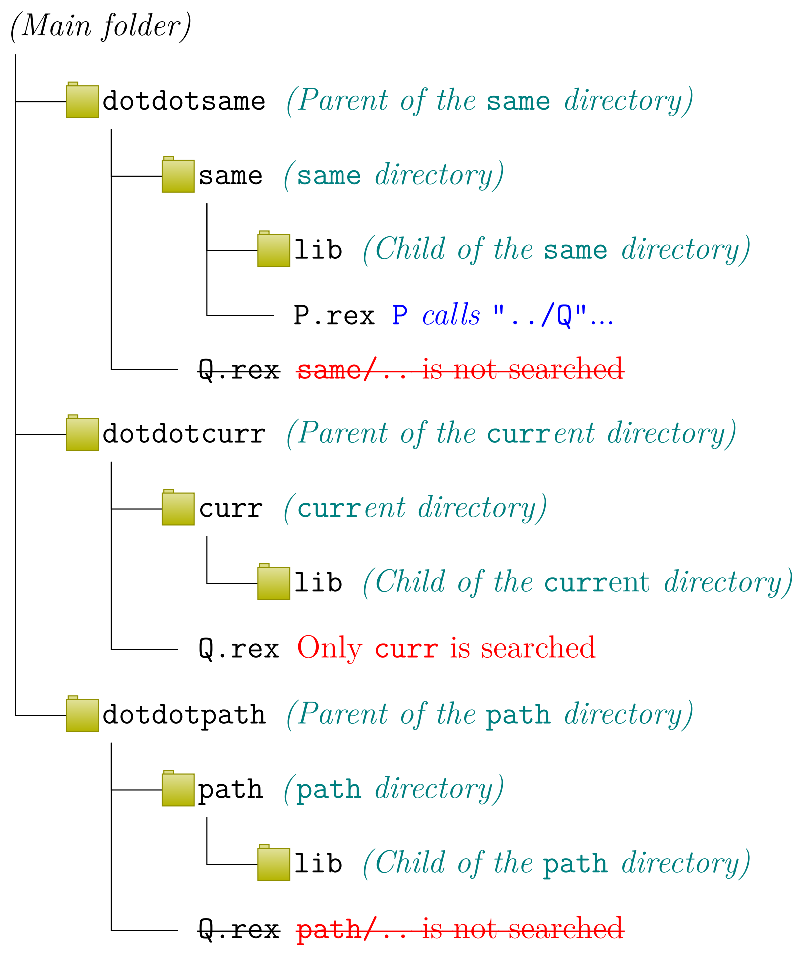
Now for the anomaly: if instead of searching in a *child* directory we want to search in a *parent* directory, we should be able to write `Call "../Q"` (or `Call "..\Q"` under Windows or OS/2), and expect that the search algorithm searches for `Q` first in the *parent* directory of `same`, then in the *parent* directory of `curr`, and finally in the *parent* directory of `path`. In the same way that `"lib/Q"` was *relative* to `same`, `curr` and `path`, in that order, we would expect that `"../Q"` would be, similarly, *relative* to `same`, `curr` and `path`, in that order: `Q` should be searched for in `"same/.."`, `"curr/.."` and `"path/.."`. Is this what is going to happen?

Let us draw the corresponding directory structure diagram before answering this question. This is what we should be expecting to happen.


(Main folder)
dotdotsame (Parent of the same directory)
same (same directory)
lib (Child of the same directory)
P.rex P calls "../Q"...
Q.rex ① ...Q can be located here, ...
dotdotcurr (Parent of the current directory)
curr (current directory)
lib (Child of the current directory)
Q.rex ② ...or here, ...
dotdotpath (Parent of the path directory)
path (path directory)
lib (Child of the path directory)
Q.rex ③ ...or here.

What is really happening, when we `Call "../Q"`? That, instead of searching in `same`, `curr` and `path`, *only* `curr` *is searched*: the search is *limited to the current directory*. This is the anomaly.[3]

[3]Indeed, the anomaly also applies when we `Call "./Q"`, but, since this should be equivalent to `Call Q`, which works perfectly, we have considered the `Call "../Q"` case to be more representative.


(Main folder)
dotdotsame (Parent of the same directory)
same (same directory)
lib (Child of the same directory)
P.rex P calls "../Q"...
~~Q.rex~~ ~~same/.. is not searched~~
dotdotcurr (Parent of the current directory)
curr (current directory)
lib (Child of the current directory)
Q.rex Only curr is searched
dotdotpath (Parent of the path directory)
path (path directory)
lib (Child of the path directory)
~~Q.rex~~ ~~path/.. is not searched~~

I have to say that this anomaly, apart from being contrary to the documentation, is something that is *highly unexpected*. Two ooRexx committers believed that ooRexx worked as one would expect and as it is documented, not as it really works.

2 A technical explanation

What is going on, what is the cause of this anomaly? By looking at the source code,[4] one can find, in `SysFileSystem.cpp`, a platform-dependent file, a boolean function called `primitiveSearchName`. This function, in case the checked *filename* verifies `hasDirectory(filename)`, *bypasses the complete*

[4]When I started this investigation, my familiarity with the ooRexx source code was zero. I have to thank Erich Steinböck for the initial pointers to the source.

search and limits it to the current directory.

2.1 In the Unix-like side of things

And what is the code for this `hasDirectory` function? It is slightly different for Windows and for Unix-like. Here's the Unix-like version:

```
/**
 * Test if a filename has a directory portion
 *
 * @param name   The name to check.
 *
 * @return true if a directory was found on the file, false if
 *         there is no directory.
 */
bool SysFileSystem::hasDirectory(const char *name)
{
    // hasDirectory() means we have enough absolute directory
    // information at the beginning to bypass performing path searches.
    return name[0] == '~' || name[0] == '/' ||
           (name[0] == '.' && name[1] == '/') ||
           (name[0] == '.' && name[1] == '.' && name[2] == '/');
}
```

A filename that starts with the `'~'` character resides in a *home* directory (i.e., in a directory of the form `/user/username`), and, therefore, it is absolute. Similarly, a filename that starts with the `'/'` character is also absolute, as `'/'` is the *root* directory. And, of course, when a filename is absolute, it does not make any sense to search in `same`, `curr`, `path`, or anywhere else: the filename itself contains enough information to unequivocally locate the file, and therefore it is rational and adequate to skip the search order altogether.

On the other hand, when a filename starts with `'./'` or `'../'`, it is *not* (contrary to the function documentation) absolute, but *relative*. Relative to what? Relative to the directory we are considering. `'.'` is a no-op, so that, for example, `'./lib'`, relative to `'same'`, means `'same/./lib'`, which is equivalent to `'same/lib'`.

Similarly, `'..'` means *the parent directory* of the considered directory, so that, for example, `'../file'`, relative to `'same'`, means `'same/../file'`, which, assuming that `'dotdotsame'` is the parent directory of `'same'`, would be equivalent to `'dotdotsame/file'`.

Indeed, the last two lines,

```
(name[0] == '.' && name[1] == '/') ||
(name[0] == '.' && name[1] == '.' && name[2] == '/');
```

can be eliminated,[5] and then the interpreter happily accepts all filenames starting with `'./'` or `'../'`, and checks them against all the directories (i.e., against `'same'`, `'curr'` and `'path'`, in our simplified version).

It works identically, in that respect, to other language processors, like the GNU C/C++ compiler, `gcc`, or the Microsoft Visual Studio C/C++ compiler, `cl`, where constructions like `'.'` and `'..'` are used all the time, for example in `#include` directives, and they are resolved against *all the directories* specified using the `-I` compiler option (or using some other compiler mechanism).

In fact, one gets the impression that these two lines are not part of the original code, but were added later, as an afterthought. This idea is, of course, speculation, but it is backed by the fact that the OS/2 version of Object Rexx searches in `curr` *and* in `path` (it cannot search in `same`, because Object Rexx for OS/2 does not have the concept of the "same" directory).

Still more: if these two lines are removed, the comments in the source code cease to be wrong.

2.2 In the Windows side of things

Let us now take a look at the Windows side of things to see if we can get more information. Here is the Windows version of `hasDirectory`:

```
/**
 * Test if a filename has a directory portion
 *
 * @param name   The name to check.
 *
 * @return true if a directory was found on the file, false if
 *         there is no directory.
 */
bool SysFileSystem::hasDirectory(const char *name)
{
    // hasDirectory() means we have enough absolute directory
    // information at the beginning to bypass performing path searches.
    // (there are more ways to specify a drive than just d: but still ..)
    return name[0] == '\\' || name[1] == ':' ||
          (name[0] == '.' && name[1] == '\\') ||
          (name[0] == '.' && name[1] == '.' && name[2] == '\\');
}
```

[5]As a proof-of-concept, I have submitted a patch (see the attachment named `unix.SysFileSystem.cpp.diff` in SourceForge bug no. 1865, `https://sourceforge.net/p/oorexx/bugs/1865/`) that eliminates them. After applying the patch, the full ooRexx test suite passes.

The code is very similar to the Unix-like version, with the necessary adjustments, i.e., substituting `"/"` by `"\"`, or handling the existence of drives.

> Indeed, `name[1] == ':'` is not exactly absolute, since one can use forms like `D:path\file`, which are *relative to the current directory of the* `D:` *drive*; similarly, `name[0] == '\\'` is not absolute either, but *relative to the root of the current drive*. The `"."` and `".."` cases are identical to the Unix-like version.

What makes the difference between the Windows version and the Unix-like version of the interpreters is not the `hasDirectory` function, but how the "normal" cases are handled (i.e., those were `hasDirectory` is `false`).

I will try to summarize how the interpreter works. In both versions, a super-path is formed first. It contains the `same` directory, the current directory (in the form `"."`), the value of the application-defined extra path, if any, and, finally, the values of the `REXX_PATH` and `PATH` environment variables.

The Unix-like version takes this super-path, breaks it into directories, and then concatenates each of these directories to the supplied filename,[6] to obtain (hopefully) absolute filenames.

The Windows version, on the other hand, takes the constructed super-path and the supplied filename and passes them (along with the corresponding extension, if appropriate) as arguments to the Windows `SearchPath` API. `SearchPath` performs the search and, if a file is found, it returns its path and file name.

At this point we find something: `SearchPath` *does not work as expected on filenames that start with* `".\"` *or* `"..\"`. The developers were well aware of this fact: see the comments before the `hasDirectory` call in the Windows version of `SysFileSystem::primitiveSearchName`:

```
// if this appears to be a fully qualified name, then check it as-is and
// quit.  The path searches might give incorrect results if performed with
// such a name and this should only check on the raw name.
if (hasDirectory(tempName))
{
    // check the file as is first
    return checkCurrentFile(tempName, resolvedName);
}
```

Let us forget about the idea that filenames starting with `".\"` or `"..\"` are "fully qualified" — we have already discussed it before. The point is that

[6]We do not take into consideration the handling of file extensions at this point of our reasoning.

the developers know that `SearchPath` does not give “correct” results, and *are programming against this fact.*

One gets the impression —but we are, once more, speculating— that the precautions that are necessary in Windows were later backported to the Unix-like world, so that the interpreter worked in the exact same way under Windows and under Unix-like systems.

> The fact is that getting the Windows version to work “as expected” (i.e., as described in the manual) would be more involved than simply suppressing two lines in `hasDirectory`: one would need to manually form each and every filename, by concatenating the different directories contained in the super-path to the supplied filename, as in the Unix-like version. But the Windows world is much more complicated, in this respect, than the Unix-like world. In fact, it is a really *horrible* world.[7]

3 How to interpret the anomaly

3.1 As an interpreter bug

How to interpret this anomaly? What we know for sure is that there is a *discrepancy* between the documentation and the observed behaviour of the interpreter.

The first, and maybe the more obvious, possibility, would be to stipulate that the anomaly is an *interpreter bug*. The behaviour of the interpreter should be corrected, so that it works as documented. Although this might seem the easiest and more reasonable path to follow, implementing it would have three disadvantages.

First, it would change the established behaviour of the interpreter. There might conceivably exist some programs that rely on the current, undocumented, behaviour, and patching this behaviour might break these programs.

Second, it would force some users to retrain their conceptions. Some programmers, for example, believe that `'.'` has to refer to the *current* directory. But eliminating the anomaly would mean that `Call './program'` would search for `program` in the `same` directory, in the `curr`ent directory and in the `path` directories, instead of only in the current directory, which may seem difficult to accept to these programmers.

[7]See, for example, the following Microsoft article, *File path formats on Windows systems* (`https://learn.microsoft.com/en-us/dotnet/standard/io/file-path-formats`), where one may find filenames starting with `"\\.\"` or `"\\?\"`, apart from UNC names, like `\\server\share\path-to\file`, and so on.

Third, ironing out the anomaly needs a substantial amount of programming, in the Windows side of things.

3.2 As a documentation bug

The second possibility would be to stipulate that, instead of constituting an interpreter bug, the anomaly really is a *documentation bug*, i.e., that the documentation should be patched, not the interpreter. The anomaly would not be a bug, but an (undocumented) feature. Patching the documentation would be much more economical than patching the interpreter, of course, but it also carries its own set of disadvantages.

First, and more important, it is difficult to explain to the user: "in most cases, the same, current and path directories will be searched, but, look, when the filename starts with `'.'` or `'..'`, well, we then have another set of rules...". We would be introducing an asymmetry without giving, at the same time, a convincing explanation for that asymmetry.

Second, it *limits* the possibilities of the user, instead of giving her maximal freedom and allow her to limit herself is she so desires. Additionally, this limitation is almost *unfixable* (more about that below, p. 16).

Third, it introduces rules that are *internally inconsistent.*

As these two last points are important, they will get separate elaboration.

3.3 A limitation that is very difficult to fix

If we want `Call "../program"` to look first in the parent of the same directory, we can always get the value of this directory (using a `Parse Source` instruction), temporarily change the current directory to the same directory, and then issue our `Call` instruction. This is an ugly hack, but it works.

But —and this is a big "but"—, when instead of a `Call` instruction what we have is a `::Requires` directive, *there is no reasonable way to fix the behaviour of* `::Requires`, because directives are processed *before* any code in the program has had any opportunity to run. We could attempt to use a security manager to intercept and reroute the target of `::Requires` directives; unfortunately, the security manager is broken for `::Requires`.[8]

3.4 An internal inconsistency

Indeed, there *is* a certain way to overcome the effects of the anomaly, although it is such an ugly kludge that we have refrained, until now, from

[8]See SourceForge bug #1886 (`https://sourceforge.net/p/oorexx/bugs/1886/`).

mentioning it: where `Call "../program"` (or `::Requires "../program"`) does not work as intended, substitute it by `Call "lib/../../program"` (or `::Requires "lib/../../program"`).[9] The fact that the expressions `"../program"` and `"lib/../../program"` are supposed to be equivalent but they still produce different effects point to an unfortunate internal inconsistency in the handling of filenames (as we will see later, this internal inconsistency is a (mis-)feature of the Windows `SearchPath` API).

3.5 Elements for a decision

We have some arguments that lead us to interpret our anomaly as a language processor bug, and some other arguments that lead us to interpret it as a documentation bug. Each set of arguments also entail their own set of disadvantages.

How to come to a decision? One additional argument would run as follows: maybe we are lacking some information; maybe there is a “Rexx way” of doing things, of which we are not aware enough; and maybe, if we studied how other interpreters tackle the same problem (and even how other environments tackle the same problem) we would, once in possession of a wider perspective, see things differently, and the decision would come to us naturally, in the light of our new understandings.

4 A number of tests

4.1 Interpreters and operating systems

To study how other interpreters tackle the Search Order problem, we have written a standardized test suite (`"sotest.rex"`) that works across several operating systems and under different interpreters. The suite, written in Classic Rexx, has been tested under three operating systems:

- OS/2 — OS/2 4.52 (ArcaOS 5.0.7).
- Ubuntu — Ubuntu 22.04.01 LTS.
- Windows — Windows 11 Pro (10.0.22621.1413).

and against four different interpreters (we have checked two versions of ooRexx for reasons that will be explained below[10]):

[9] Under some Unix-like systems, the `"lib"` directory has to exist; Windows does not check for its existence, i.e., it treats constructs like the above in a purely *syntactical* way.

[10] In section 4.2.2, titled *The* `hasExtension` *bug*, on page 19.

- OS/2 Procedures Language 2/REXX ("Classic Rexx") (REXXSAA 4.00 3 Feb 1999).
- Regina Rexx (REXX-Regina_3.9.5 5.00 25 Jun 2022) under OS/2, Ubuntu and Windows.
- Object Rexx for OS/2 (OBJREXX 6.00 18 May 1999).
- Open Object Rexx 5.0.0 (REXX-ooRexx_5.0.0(MT)_64-bit 6.05 23 Dec 2022) under Ubuntu and Windows.
- Open Object Rexx 5.1.0 beta r12651 (REXX-ooRexx_5.1.0(MT)_64-bit 6.05 10 Mar 2023) under Windows.

4.2 Two bugs in two interpreters

In the process of running the tests, we uncovered two bugs in two different interpreters.

4.2.1 The SAA bug

The first bug was found in the REXXSAA ("Classic") Rexx interpreter under OS/2. `REXX.INF`, the accompanying help file for REXXSAA, reads "REXX searches for external functions in the following order: (...) 3. REXX functions in the current directory, with the current extension[;] 4. REXX functions along environment PATH, with the current extension (...)".

However, this does not seem to work as documented, as one can quickly check with a very simple test: the current extension is never checked, only the default one. We will refer to this bug as "the SAA bug".

> REXX searches for external functions in the following order:
>
> 1. Functions that have been loaded into the macrospace for pre-order execution
> 2. Functions that are part of a function package.
> 3. REXX functions in the current directory, with the current extension
> 4. REXX functions along environment `PATH`, with the current extension
> 5. REXX functions in the current directory, with the default extension
> 6. REXX functions along environment `PATH`, with the default extension
> 7. Functions that have been loaded into the macrospace for post-order execution.

> [`REXX.INF` for the `REXXSAA` OS/2 interpreter (v. 4.00 3 Feb 1999)]

When collecting and comparing our test results, we will amend the test results for REXXSAA under OS/2 as if the interpreter worked as documented. This will allow us to unveil a number of interesting coincidences that would not surface otherwise.

4.2.2 The `hasExtension` bug

The second bug was found in the Windows version of the ooRexx interpreter (v. 5.0.0). Here is the relevant fragment of the ooRexx Reference manual:

> The second element of the search process is the file extension. If the routine name contains at least one period, then this routine is extension qualified. The search locations above will be checked for the target file unchanged, and no additional steps will be taken. If the routine name is not extension qualified, then additional searches will be performed by adding file extensions to the name.
>
> [*Open Object Rexx Reference* (rexxref.pdf), release 5.0.0, section 7.2.1.1]

The interpreter decides whether a routine "is extension qualified" by calling (the Windows version of) the boolean `SysFileSystem::hasExtension` routine, reproduced below.

```
/**
 * Test if a filename has an extension.
 *
 * @param name   The name to check.
 *
 * @return true if an extension was found on the file, false if there
 *         is no extension.
 */
bool SysFileSystem::hasExtension(const char *name)
{
    const char *endPtr = name + strlen(name) - 1;

    // scan backwards looking for a directory delimiter.  This name should
    // be fully qualified, so we don't have to deal with drive letters
    while (name < endPtr)
    {
        // find the first directory element?
        if (*endPtr == '/')
        {
            return false;        // found a directory portion before an extension...
            // we're extensionless
        }
        // is this the extension dot?
        else if (*endPtr == '.')
        {
            // return everything from the period on.  Keeping the period on is a convenience.
            return true;
        }
        endPtr--;
    }
    return false;          // not available
}
```

But `SysFileSystem::hasExtension` has a typo: it checks backwards for `'/'`, the Unix-like separator character, instead of checking for `'\'`, the Windows separator character. This opens three possibilities:

- The filename has no extension and no directory contains a dot (`'.'`): `hasExtension` will run to its end, and return `false` (i.e., "filename has no extension"), which is correct.
- The filename has an extension (i.e., it contains a dot): `hasExtension` will find the dot, and return `true` (i.e., "filename has an extension"), which is correct.
- The filename has no extension, but one of the directories in the file path contains a dot: in this case, `hasExtension` will find the dot, and return `true` (i.e., "filename has an extension"), which is *not* correct.

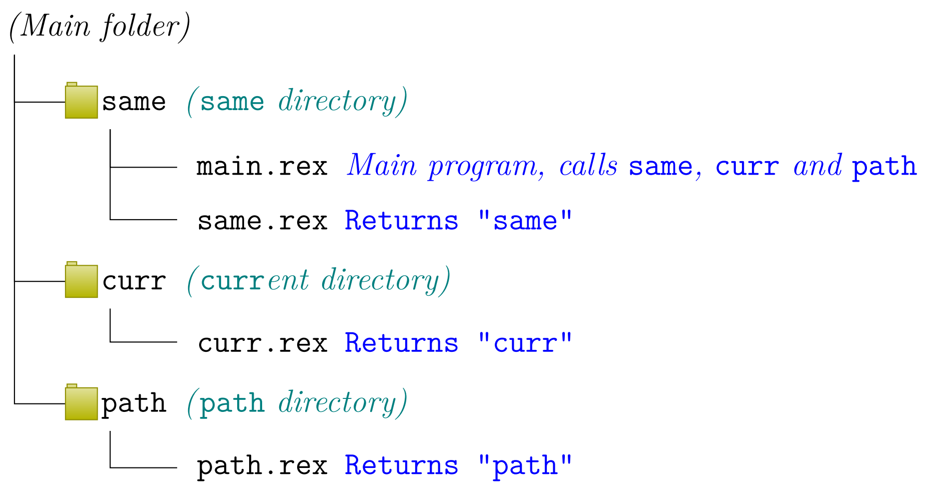
As it can be seen, the bug is difficult to trigger (one needs a filename of the form `"my.dir\name`); this is probably the reason why it was not detected previously.[11] We will refer to this bug as "the `hasExtension` bug".

4.3 The directory structure

The test suite uses the simplified directory structure we have described previously, and places a test program in each and every one of the searchable places, to see if the corresponding search algorithm is able to find them or not.

As a starting example, let us go back, for a moment, to the simple calls example (p. 7). Since the interpreter is supposed to be able to search in the `same`, `curr` and `path` directories, we can put a simple program in each of these places, and call every one of them in turn.

[11] I reported this bug (`https://sourceforge.net/p/oorexx/bugs/1870/`), attached a patch and uploaded an updated test case. The updated version of `hasExtension` is part of the 5.1.0 beta since commit r12651 (`https://sourceforge.net/p/oorexx/code-0/12651/`).



For example, in this simplified directory structure, `main` will first call `same`, then `curr` and then `path`. These programs will really be very simple, they will just return their own name (without the `".rex"` extension). The test program will then only have to check the returned value. When a program is not found, the corresponding syntax condition will be trapped by a `Signal On` trap and handled accordingly.

The full directory structure is depicted in a diagram that can be found on the following page. We have added an intermediate subdirectory called `"subdir"` for future expansion.[12]

[12]We will not be using this directory for Rexx tests, but it will come in handy when testing the features of the Visual Studio C/C++ `#include` directive.

```
(Main folder)
├── sotest.rex (Calls subdir/dotdotsame/same/main.rex)
└── subdir (For future expansion)
    ├── dotdotsame (Parent of the same directory)
    │   ├── dotdotsame.rex (Returns "dotdotsame")
    │   └── same (same directory)
    │       ├── main.rex (Main program)
    │       ├── same.rex (Returns "same")
    │       └── lib (Child of the same directory)
    │           └── samelib.rex (Returns "samelib")
    ├── dotdotcurr (Parent of the current directory)
    │   ├── dotdotcurr.rex (Returns "dotdotcurr")
    │   └── curr (current directory)
    │       ├── curr.rex (Returns "curr")
    │       └── lib (Child of the current directory)
    │           └── currlib.rex (Returns "currlib")
    └── dotdotpath (Parent of the path directory)
        ├── dotdotpath.rex (Returns "dotdotpath")
        └── path (path directory)
            ├── path.rex (Returns "path")
            └── lib (Child of the path directory)
                └── pathlib.rex (Returns "pathlib")
```

Directory structure for `sotest.rex`

The test initiator program, `sotest.rex`, resides in the main folder. After some trivial housekeeping, it calls the main program, `main.rex`, located in the `"subdir/dotdotsame/same"` subdirectory. `Main.rex` then sets the current directory to `"subdir/dotdotcurr/curr"`, and, correspondingly, the path to `"subdir/dotdotpath/path"`. Every program will be first called without specifying the `".rex"` extension, and then it will be called again with that extension: this has allowed us to unveil the SAA and the `hasExtension` bugs (see pp. 18ff).

4.4 Types of tests

4.4.1 Common tests

The first 30 tests are common to all the operating systems and interpreters, and to some other environments, like `CMD.EXE` under Windows and the Windows `SearchPath` API. They can be collected in five groups:

Simple calls:

1. `Call "same"`,
2. `Call "same.rex"`,
3. `Call "curr"`,
4. `Call "curr.rex"`,
5. `Call "path"`, and
6. `Call "path.rex"`.

Downwards-relative calls:

7. `Call "lib/same"`,
8. `Call "lib/same.rex"`,
9. `Call "lib/curr"`,
10. `Call "lib/curr.rex"`,
11. `Call "lib/path"`, and
12. `Call "lib/path.rex"`.

Dot-relative calls:

13. `Call "./same"`,
14. `Call "./same.rex"`,
15. `Call "./curr"`,
16. `Call "./curr.rex"`,

17. `Call "./path"`, and
18. `Call "./path.rex"`.

Upwards-relative calls:

19. `Call "../same"`,
20. `Call "../same.rex"`,
21. `Call "../curr"`,
22. `Call "../curr.rex"`,
23. `Call "../path"`, and
24. `Call "../path.rex"`.

Upwards-relative calls with a trick:[13]

25. `Call "lib/../../same"`,
26. `Call "lib/../../same.rex"`,
27. `Call "lib/../../curr"`,
28. `Call "lib/../../curr.rex"`,
29. `Call "lib/../../path"`, and
30. `Call "lib/../../path.rex"`.

4.4.2 Drive-relative tests

Windows and OS/2 have *drives* and *drive letters*. This will allow us to prepare and run some extra tests.

Under Windows, our test program will be able to use the `SUBST` command to temporarily assign new drive letters to our current and path directories, and then change the current directory and the path to point to these new drives.

OS/2 does not have a `SUBST` command, so that we will have to manually assign the new drive letters by a mechanism external to our test program, and then list these drives (without the colons) in the `SOTEST_DRIVES` environment variable.

Backslash-relative calls (`myDir` is the directory where `main.rex` is located, without the drive letter):

31. `Call (myDir"same")`,

[13] *Cfr.* section 3.4, *An internal inconsistency*, on p. 16.

32. `Call (myDir"same.rex")`,
33. `Call "\dotdotcurr"`,
34. `Call "\dotdotcurr.rex"`,
35. `Call "\dotdotpath"`, and
36. `Call "\dotdotpath.rex"`.

The idea behind backslash-relative calls is the following: if, in many contexts, `"\path\filename"` is supposed to be *relative to the current directory* and ooRexx should check the "same" directory before the current directory, then `"\path\filename"` should be first checked against the drive of the same directory, then against the drive of the current directory, and then, in turn, against every one of the drives found in the different directories specified in the paths.

Currently, no Rexx interpreter goes beyond checking the current directory, but Python's `pathlib` module, for example, has an (implied) concept of backslash-relative filenames which allows this kind of extended checking (see section 4.5.4, titled *Python* `pathlib` *and backslash-relative filenames*, below, on page 27).

Letter-relative calls (assume that `D:` is the `same` directory drive, `X:` is the current directory drive, and `Y:` is the path directory drive):

37. `Call "D:lib\samelib"`,
38. `Call "D:lib\samelib.rex`,
39. `Call "X:curr\curr"`,
40. `Call "X:curr\curr.rex"`,
41. `Call "Y:path\path"`, and
42. `Call "Y:path\path.rex"`.

Similarly to backslash-relative calls, the main idea of letter-relative calls is the following: we already know that, under Windows and OS/2, each drive has its own current directory. If a particular drive, say `D:`, has a current directory of `"\dir1\dir2"` and we specify a filename as `"D:filename"`, we end up with `"D:\dir1\dir2\filename"`.

Following this logic, and if ooRexx, as per the manual, has to check first the "same" directory, one would be entitled to expect that if, say, the "same" directory was `"C:\some\path"` and one specified the filename as `"C:more\myname"`, `"C:\some\path\more\myname"` would be checked, and similarly for all the different directories specified in the paths.

Currently, no Rexx interpreter implements this behaviour.

Drive-absolute calls (`myPath` is the directory where `main.rex` is located, including the drive letter; `X:` is the current directory drive, and `Y:` is the path directory drive):

43. `Call (myPath"same")`,
44. `Call (myPath"same.rex")`,
45. `Call "X:\curr\curr"`,
46. `Call "X:\curr\curr.rex"`,
47. `Call "Y:\path\path"`, and
48. `Call "Y:\path\path.rex"`.

These calls, being absolute, always work, under all interpreters.

4.5 Special tests

In addition to testing the behaviour of a number of different Rexx interpreters under several operating systems, we have also tested, for reference and comparison purposes, several products and environments more.

4.5.1 The Windows Command Line Interpreter (`CMD.EXE`)

To test the behaviour of the Windows Command Line interpreter (`CMD.EXE`), we used a modified version of our test program. Instead of using a `Call` instruction to call our programs, we used an `Address COMMAND` instruction. This allowed us to observe the resolution algorithms employed by `CMD.EXE`.

4.5.2 The Windows `SearchPath` API

To test the behaviour of the Windows `SearchPath` API, we have written a trivial program, `SearchPath.c`, to encapsulate the API usage as an `.EXE` file.

In this case, we substitute the `Call` instruction by

```
Address COMMAND sameDir"/SearchPath" sameDir";.;"pathDir target ".rex"
```

where `target` is the file to resolve, `sameDir` is the directory where `main.rex` is located, and `pathDir` is the `path` directory.

4.5.3 Checking the C/C++ compilers

We have also checked the behaviour of the `gcc` and `cl` (Visual Studio) C/C++ compilers when handling the `#include` directive. Both of them honour the `-I`*directory* option (which can be specified multiple times), and both of them accept without any complaints all the combinations of downward-relative, upwards-relative and dot-relative filenames. Additionally, the compilers have a "same" directory concept (i.e., the directory where the file using the `#include` directive is located), which is extended, in the case of the Visual Studio compiler, in an interesting, recursive way:

> The preprocessor searches for include files in this order:
>
> 1. In the same directory as the file that contains the `#include` statement.
> 2. In the directories of the currently opened include files, in the reverse order in which they were opened. The search begins in the directory of the parent include file and continues upward through the directories of any grandparent include files.
> 3. Along the path that's specified by each `/I` compiler option.
> 4. Along the paths that are specified by the `INCLUDE` environment variable.
>
> [*Fragment of the Visual Studio compiler documentation*]

4.5.4 Python `pathlib` and backslash-relative filenames

Python's `pathlib` is a "module [that] offers classes representing filesystem paths with semantics appropriate for different operating systems". We are especially interested in the description of the slash operator. We extract the following quote:[14]

> The slash operator helps create child paths, like `os.path.join()`. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):
>
> [`pathlib` — *Object-oriented filesystem paths — Operators*]

The following example (taken from the same source) handles filenames starting with a (back)slash as relative filenames.

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

[14]See `https://docs.python.org/3/library/pathlib.html#operators`

5 Classifying and analysing the results

Our test program, `sotest.rex`, formats the results of running our test suite in such a way that the results set is, by itself, a Rexx program. This program collects the results in a stem, which is then returned when the program ends. This is very convenient, because it greatly simplifies the automation of certain tasks, for example, the comparison of different result sets.

As an example, the following program listing is the output produced by running `sotest.rex` using the ooRexx interpreter (5.1.0 beta, r12651) under Windows 11 (see test *Open Object Rexx 5.1.0 beta after commit r12651 under Windows* on page 47).

```
/******************************************************************************
  sotest.rex -- A Search Order test suite

  Interpreter:      REXX-ooRexx_5.1.0(MT)_64-bit 6.05 10 Mar 2023
  Operating system: WindowsNT
  Full name:        D:\Dropbox\ooRexx\sotest\sotest.rex
  Main routine:     D:\Dropbox\ooRexx\sotest\subdir\dotdotsame\same\main.rex

  Test suite starting on 17 Mar 2023 at 11:22:16

  The following values have been set:

  Same directory:    'D:\Dropbox\ooRexx\sotest\subdir\dotdotsame\same'
  Current directory: 'D:\Dropbox\ooRexx\sotest\subdir\dotdotcurr\curr'
  Path:              'D:\Dropbox\ooRexx\sotest\subdir\dotdotpath\path'

  This is ooRexx. Search order is extension-first
 ******************************************************************************/
Pass.1  = .true;  Pass.1.test  = 'same'
Pass.2  = .true;  Pass.2.test  = 'same.rex'
Pass.3  = .true;  Pass.3.test  = 'curr'
Pass.4  = .true;  Pass.4.test  = 'curr.rex'
Pass.5  = .true;  Pass.5.test  = 'path'
Pass.6  = .true;  Pass.6.test  = 'path.rex'
Pass.7  = .true;  Pass.7.test  = 'lib\samelib'
Pass.8  = .true;  Pass.8.test  = 'lib\samelib.rex'
Pass.9  = .true;  Pass.9.test  = 'lib\currlib'
Pass.10 = .true;  Pass.10.test = 'lib\currlib.rex'
Pass.11 = .true;  Pass.11.test = 'lib\pathlib'
Pass.12 = .true;  Pass.12.test = 'lib\pathlib.rex'
Pass.13 = .false; Pass.13.test = '.\same'
Pass.14 = .false; Pass.14.test = '.\same.rex'
Pass.15 = .true;  Pass.15.test = '.\curr'
Pass.16 = .true;  Pass.16.test = '.\curr.rex'
Pass.17 = .false; Pass.17.test = '.\path'
Pass.18 = .false; Pass.18.test = '.\path.rex'
Pass.19 = .false; Pass.19.test = '..\dotdotsame'
Pass.20 = .false; Pass.20.test = '..\dotdotsame.rex'
Pass.21 = .true;  Pass.21.test = '..\dotdotcurr'
Pass.22 = .true;  Pass.22.test = '..\dotdotcurr.rex'
Pass.23 = .false; Pass.23.test = '..\dotdotpath'
Pass.24 = .false; Pass.24.test = '..\dotdotpath.rex'
Pass.25 = .true;  Pass.25.test = 'lib\..\..\dotdotsame'
Pass.26 = .true;  Pass.26.test = 'lib\..\..\dotdotsame.rex'
Pass.27 = .true;  Pass.27.test = 'lib\..\..\dotdotcurr'
Pass.28 = .true;  Pass.28.test = 'lib\..\..\dotdotcurr.rex'
Pass.29 = .true;  Pass.29.test = 'lib\..\..\dotdotpath'
Pass.30 = .true;  Pass.30.test = 'lib\..\..\dotdotpath.rex'
/* Executing 'SUBST Z: D:\Dropbox\ooRexx\sotest\subdir\dotdotcurr'             */
/* Executing 'SUBST Y: D:\Dropbox\ooRexx\sotest\subdir\dotdotpath'             */
/* Changing current directory to Z:\                                           */
/* Changing PATH to Y:\                                                        */
Pass.31 = .false; Pass.31.test = '\Dropbox\ooRexx\sotest\subdir\dotdotsame\same\same'
Pass.32 = .false; Pass.32.test = '\Dropbox\ooRexx\sotest\subdir\dotdotsame\same\same.rex'
Pass.33 = .true;  Pass.33.test = '\dotdotcurr'
Pass.34 = .true;  Pass.34.test = '\dotdotcurr.rex'
Pass.35 = .false; Pass.35.test = '\dotdotpath'
Pass.36 = .false; Pass.36.test = '\dotdotpath.rex'
Pass.37 = .false; Pass.37.test = 'D:lib\samelib'
Pass.38 = .false; Pass.38.test = 'D:lib\samelib.rex'
Pass.39 = .true;  Pass.39.test = 'Z:curr\curr'
Pass.40 = .true;  Pass.40.test = 'Z:curr\curr.rex'
Pass.41 = .true;  Pass.41.test = 'Y:path\path'
Pass.42 = .true;  Pass.42.test = 'Y:path\path.rex'
Pass.43 = .true;  Pass.43.test = 'D:\Dropbox\ooRexx\sotest\subdir\dotdotsame\same\same'
Pass.44 = .true;  Pass.44.test = 'D:\Dropbox\ooRexx\sotest\subdir\dotdotsame\same\same.rex'
Pass.45 = .true;  Pass.45.test = 'Z:\curr\curr'
Pass.46 = .true;  Pass.46.test = 'Z:\curr\curr.rex'
Pass.47 = .true;  Pass.47.test = 'Y:\path\path'
Pass.48 = .true;  Pass.48.test = 'Y:\path\path.rex'
Pass.0 = 48
Return Pass.
```

5.1 Eliminating extensions

We will now run a short program called "`noextension.rex`" against the whole set of our test results. The program has to be located in the a directory where all the test results reside, and it detects which result sets are extension-dependent (i.e., which tests show different results for `Call "program"` and for `Call "program.rex"`).

Running this program once allows us to be sure that all our test results are extension-independent, except for `os2.rexxsaa` (see p. 43), which exhibits the SAA bug and we will substitute for its manually amended version, `os2.rexxsaa.fixed` (see p. 42), and for `windows.oorexx-5.0.0` (see p. 46), which exhibits the `hasExtension` bug, and will be substituted by the patched 5.1.0 beta version of the test, `windows.oorexx-5.1.0-beta-r12651` (see p. 47).

5.2 Equivalence classes

Some pairs of result sets are identical modulo the tests they have in common,[15] in the sense that the same tests pass or fail for the first and the second result sets, and some other pairs are not identical. We have written a test program, `"compare.rex"` that compares two result sets and tells us if they are identical or not.

Identity is an equivalence relation: we will thus be able to construct a set of equivalence classes.

5.2.1 Class 1: Regina, REXXSAA (amended), CMD.EXE

Using `compare.rex`, we can verify that our result sets for Regina Rexx under Windows (`windows.regina`, see p. 48), OS/2 (`os2.regina`, see p. 40) and Ubuntu (`ubuntu.regina`, see p. 45) are equivalent.

Additionally, these are equivalent to the amended result set for the OS/2 REXXSAA interpreter,[16] (`os2.rexxsaa.fixed`, see p. 42), and to the result set of the special `CMD.EXE` test (`windows.cmd`, see p. 49).

Neither of these interpreters and environments use the "same" or (caller's) directory, so that all the `same` tests will fail. Additionally, the members of this equivalence class are the most restrictive of all three. We will find a nice description of these restrictions in *The Regina Rexx Interpreter*, the manual for version 3.9.5 of Regina.

> **1.4.2 External Rexx programs**
>
> [...]
>
> When processing an environment variable, the content is split into the different paths and each path is processed separately. Note that the search algorithm to this point is ignored if the program name contains a file path specification. eg. if `"CALL .\MYPROG"` is called, then no searching of `REGINA_MACROS` or `PATH` is done; only the concatenation of suffixes is carried out.
>
> [*The Regina Rexx Interpreter* (regina.pdf), version 3.9.5, section 1.4.2.]

[15] Remember that in Unix-like systems we run 30 tests, but in systems that have drive letters the number of tests is 48.

[16] That is, the one with the SAA bug fixed.

"If the program name contains a file path specification" is implemented in a very simple way: a check for a separator character (i.e., `"\"` under Windows or OS/2 and `"/"` under Unix-like systems) is run against the program name.[17]

5.2.2 Class 2: Object Rexx for OS/2

The result set for Object Rexx for OS/2 (`os2.objrexx`, see p. 41) is a singleton, i.e., it forms an equivalence class by itself, since no other result sets are equivalent to it.

Object Rexx does not have a concept of a "same" directory, but it works happily with filenames that start with one or two dots followed by a backslash: it checks these filenames against the current directory, and then against all the directories specified in the `PATH` environment variable.

5.2.3 Class 3: ooRexx (5.1.0 beta) and Windows `SearchPath`

The third equivalence class is formed by the ooRexx result sets under Windows (`windows.oorexx-5.1.0-beta-r12651`, see p. 47) and under Ubuntu (`ubuntu.oorexx`, see p. 44), and the Windows `SearchPath` API (`windows.searchpath`, see p. 50).

These tests have a notion of the "same" directory, but, compared to Object Rexx, they backpedal: when a filename starts with one or two dots followed by a separator, only the current directory is searched.

5.3 In summary

Class 1, that is, Regina, Classic Rexx and the `CMD.EXE` command line interpreter, offer the most restrictive of all the behaviours we have tested. This should not be surprising, since Classic Rexx was presented, at the moment under the name "OS/2 Procedures Language 2/REXX", and it was described as follows:[18]

> **Brief Description of the REstructured eXtended eXecutor Language**

[17]See `files.c`, routine `get_external_routine`, and `configur.h` in the Regina source code.

[18]*Procedures Language/2. REXX reference*, S10G-6268-00, dated December 1991, is part of the OS/2 2.0 Technical Library. A copy can be found in the `https://github.com/RexxLA/rexx-repository/tree/master/ARB/standards/historic/references/rexx` directory.

> The REstructured eXtended eXecutor (REXX) language is a language particularly suitable for:
>
> - Command procedures
> - Application front ends
> - User-defined macros (such as editor subcommands)
> - Prototyping
> - Personal computing.
>
> [*Procedures Language/2. REXX reference*, version 2.00, p. 2-1]

The first item in the list above is "Command procedures", and there is no mention of the possibility of developing serious applications — even when quite big servers like `LISTSERV` (in `BITNET`), `NETSERV` (in `EARN`) or `TOOLS` (in `VNET`) had already been in widespread use. If the focus is centred on "command procedures", it should not be surprising that some aspects of the behaviour of the Classic Rexx interpreter are modelled against the behaviour of the command line interpreters. Regina, which has a long history, is also following, in this respect, the behaviour of the Classic Rexx interpreter.

Class 2, that is, Object Rexx for OS/2, does not have the concept of a "same" directory, but has no restrictions in the form of the filenames in accepts for `Call` or `::Requires`. In some sense, it is the most advanced of all the tested interpreters. Object Rexx is a much more mature language than Classic Rexx, and with its SmallTalk-like image, first-level classes, objects, and message sending, it is clearly conceived as a means to develop large, professional applications. Unsurprisingly, it then borrows concepts found in other programming environments: except for the absence of the "same" directory concept, its external file name resolution is the same used by the usual C/C++ compilers.

Class 3, that is, ooRexx and the Windows `SearchPath` API, introduces the concept of the "same" directory, but at the same time it back-pedals, with respect to Object Rexx, in that it has a more restrictive view of the external file name resolution (that is just the *anomaly* that started our research).

In retrospect, our hopes that we would find a "Rexx way" of handling the external search problem (see section 3.5, *Elements for a decision*, on p. 17) seem to have been unfounded. There is no "Rexx way", but nonetheless a certain trend insinuates itself: the consideration of Rexx slowly mutates, historically, starting from a "command procedure" paradigm, and tending towards a more mature programming language paradigm.

There is no "Rexx way", but, anyway, not everything is lost: we have learnt a great deal of things by designing and running our tests, and by referring to so many documents.

Indeed, we could *continue our research*, and try to expand, still more, our knowledge and our understanding of the external search problem. This knowledge might be very useful for new implementations and variants of Rexx; it can also be valuable information for the RexxLA Architecture Review Board (ARB) to ponder, and may help to design a future Rexx standard (Language level 7?); finally, it may give us more elements to definitively settle the anomaly.

Our next steps will consist of constructing a (programming) model of the external search order algorithms.

6 Modelling external search algorithms

We have written an experimental set of ooRexx classes that attempt to model the behaviour of all the interpreters and environments that we have tested. We will present here a high-level definition of these classes. The basic idea is that we should be able to write code similar to the following.

```
filename = "path/name"                        /* Our (probably relative) filename         */
searcher = .ExternalSearchVariant~new(args)   /* Create a search algorithm instance       */
program = searcher~search(filename)           /* Ask the instance to search for a filename */
If program~isNil Then                         /* A value of .nil indicates                */
  Raise Syntax 43.1 Array(filename)           /*   that the file was not found            */
Else Call (program) args                      /* File was found. Call it!                 */
```

Let us analyse this code fragment. First of all, we initialize an instance of a particular search algorithm called `ExternalSearchVariant`. This instance provides a `search` method that implements this particular search algorithm. Sending a `search` message to the instance will return `.nil` if the search does not succeed, or an absolute, resolved, filename if it does.

All external search classes will be subclasses of an abstract class, called `ExternalSearch`, WHICH will encapsulate the logic common to all the search algorithms.

6.1 Location-first and qualifier-first algorithms

An external search algorithm receives as its arguments a list of locations (directories, minidisks, etc.) and a list of qualifiers (extensions, filetypes, etc.).

An algorithm is *location-first* when every supplied location is completely searched in turn, that is, when the first location is checked against all the supplied qualifiers, then the second location is checked, and so on. The abstract class `LocationFirstExternalSearch` is a subclass of `ExternalSearch` that implements this behaviour. Regina Rexx, for example, uses a directory-first algorithm.

An algorithm is *qualifier-first* when every supplied qualifier is completely searched in turn, that is, when the first qualifier is checked in every of the supplied locations, then the second qualifier is checked, and so on. The abstract class class `QualifierFirstExternalSearch` is a subclass of `ExternalSearch` that implements this behaviour. ooRexx, for example, uses an extension-first algorithm.

6.2 Location-exception and qualifier-exception clauses

Every external search algorithm can define a *location-exception clause* and a *qualifier-exception clause*.

A *location-exception clause* examines the supplied locations and qualifiers, as well as the filename to search, and may decide that the filename will not be searched against all the locations, but only against a distinguished subset of these locations. For example, Regina Rexx restricts the search to the current directory when the filename contains a separator, and ooRexx restricts the search in the same way when the filename starts with one or two dots followed by a separator character.

A *qualifier-exception clause* examines the supplied locations and qualifiers, as well as the filename to search, and may decide that the filename will not be searched by adding all the qualifiers, but only a distinguished subset of these qualifiers. For example, Regina Rexx does not try to add extensions if a known extension is found, and ooRexx does not try to add extensions if the filename contains a dot.

6.3 The composition operation

At a certain point, a search algorithm is presented with a location, a filename, and a qualifier (which may be empty), and has to produce a complete, absolute filename and check whether the corresponding file exists or not. This operation, which we call *the composition operation* is not so simple as a naïve approach to the problem may suggest, due to the fact that the location may itself be relative, some filenames may be backslash-relative, or letter-relative, etc. Every composition algorithm may return one or more values to check. For example, ooRexx under Unix-like systems attempts to find a mixed case or uppercase filename as is, and then, if not found, it attempts to find the file again, after transforming the filename to lowercase.

Every external search class may implement its own, independent, composition operation.

6.4 The `ooRexxExternalSearch` class

Since ooRexx uses an extension-first algorithm, its corresponding class, `ooRexxExternalSearch`, will be a subclass of `QualifierFirstExternalSearch`.

Instances of `ooRexxExternalSearch` can be fully customized at initialization time:

```
mySearch = ooRexxExternalSearch~new(                 -
    (                                                - /* Directories,   */
        "same=<same directory>",                     -
        "current=<current directory>",               -
        "application=<application-defined path>",    - /* paths, and     */
        "rexx_path=<path>",                          -
        "path=<path>",                               -
    ),                                               -
    (                                                - /* extensions     */
        "same=<same extension>",                     -
        "application=<application-defined extensions>", -
    )                                                -
)
```

If an argument is not specified, the class provides the expected default. For example, we can force the search to proceed *as if* the current directory was, say, `/this/dir`, by providing a `current=` argument, but if no argument is provided, `ooRexxExternalSearch` will default to the actual current directory.

6.5 The `ReginaRexxExternalSearch` class

Since Regina Rexx uses a directory-first algorithm, its corresponding class, `ReginaRexxExternalSearch`, will be a subclass of `LocationFirstExternalSearch`.

Instances of `ReginaRexxExternalSearch` can be fully customized at initialization time:

```
mySearch = ReginaRexxExternalSearch~new(             -
    (                                                -
        "regina_macros=<path>",                      - /* Paths,         */
        "current=<current directory>",               - /* directories    */
        "path=<path>",                                - /* and            */
    ),                                                -
    (                                                 - /* extensions     */
        "same=<same extension>",                      -
        "regina_suffixes=<list>",                     -
    )                                                 -
)
```

As with `ooRexxExternalSearch`, if an argument is not specified, the class provides the expected default.

6.6 The `driveRelative` boolean attribute

The `ExternalSearch` class has a settable boolean attribute called `drive-Relative`, with a default value of `.false`. When `driveRelative` is `.true`, the composition operation is slightly modified, so that drive-relative filenames are resolved like in the `pathlib` Python module. This is experimental at the moment.

6.7 The `ooRexxEnhancedExternalSearch` class

The `ooRexxEnhancedExternalSearch` class fixes the anomaly by removing the checks for `".\"` and `"..\"`, and by additionally setting the value of `driveRelative` to `.true`. All 48 tests pass (under Windows) when we use this enhanced external search algorithm.

7 A pluggable external search system

Using the security manager feature of ooRexx, we have devised an experimental system of pluggable external search algorithms. This is currently implemented by the `"[]="` class method of the `ExternalSearch` class. The following code fragment illustrates the technique:

```
/* To be able to plug a security manager, we need a Routine object          */
routine = .Routine~newFile("/path/to/my/program.rex")

/* Routine will be called, but with our enhanced search order in effect     */
.ExternalSearch[routine] = .ooRexxEnhancedExternalSearch

/* Now call our routine with the appropiate parameters. Every CALL instruction in  */
/* "program.rex" will be resolved according to the Rexx Enhanced External Search   */
/* algorithm. The same is true for every program called from "program.rex" (we     */
/* install the security manager in a recursive way).                               */
routine~call(parameters)

::Requires ExternalSearch
```

If the comments and the blank lines are deleted, this is only *four* lines of code, which in turn are equivalent to

```
Call "/path/to/my/program.rex" parameters
```

but with the guarantee that all the calls will be resolved using our enhanced algorithm.

Of course instead of `ooRexxEnhancedExternalSearch` one can use any non-abstract subclass of `ExternalSearch`. This allows, for example, to run a program using the ooRexx interpreter, but resolve `Call` statements according to the Regina Rexx rules.

The security manager does not currently support the `::Requires` directive,[19] so that, unfortunately, our implementation cannot go beyond a mere proof-of-concept.

8 Further work

There are several avenues that we can pursue if we want to continue and expand our research. I will list several of these avenues below, in no particular order, and without any pretension of exhaustivity.

1. Find out what happens when *the directories* specified (in a path or otherwise) are themselves relative, e.g, `'..'` or `'lib'`. Write some tests, and show how the different interpreters, tools and environments handle these cases.
2. The drive-relative tests need more work. In particular, the drive-absolute tests (bad nomenclature, by the way, since it is self-contradictory) seem a little redundant, since they always pass, for all interpreters and operating systems. The use of the `driveRelative` attribute is a quick hack that has allowed us to get quite a lot of interesting information, but it indicates that the conceptual work about the topic is, in this respect, lacking.
3. It would be nice if the conceptual categories and programs we developed could provide elements to the Rexx Architecture Review Board of RexxLA to prepare a new standard for the Rexx language. Feedback from the ARB would be of great value.
4. Can our programs model the different Rexx for VM implementations? How about Rexx for z/OS? And Rexx for z/VSE? Are these completely

[19]See SourceForge bug #1886 (`https://sourceforge.net/p/oorexx/bugs/1886/`).

isolated worlds, or is it possible to create useful abstractions that encompass OS/2, Windows, the Unix-like systems, and these other operating systems at the same time?

5. Perfect our pluggable system to make it usable in large applications. We would first need the security manager to work as documented[20] with the `::Requires` directive.

9 Conclusions

We started with an apparently simple anomaly that seemed to promise us nothing, but, in the end, we have experienced an unexpectedly long journey, and a very enjoyable one at that. We have delved into the source code of ooRexx, where we have found the reasons for our anomaly; we have conjectured that a "Rexx way" of doing things could exist, and, in search of that hypothetical Rexx way, we have devised a nice collection of tests, run them, and analysed and compared their results.

There was, we have had to conclude, after all, no "Rexx way" of doing things, regarding the external search problem, but we have learnt a big number of things in the process of coming to that conclusion.

To further widen our perspective, and to better conceptualize the results of our tests, we implemented a rich set of ooRexx classes; they can model the behaviour of all the environments that we have been testing, and additionally they can also define new, arbitrary, external search algorithms. We presented, as an example, a sample enhanced ooRexx external search class that completely solves the anomaly and has some interesting additional properties.

We finished our investigation by demonstrating a proof-of-concept implementation for a pluggable external search system.

And that's it. For the moment. As we already said, above: this has been a very pleasant journey.

Tregurà de Dalt-Barcelona, February 14–May 12 2023

[20]See bugs no. 1885 (`https://sourceforge.net/p/oorexx/bugs/1885/`) and 1886 (`https://sourceforge.net/p/oorexx/bugs/1886/`).

Appendices: Test results

The tables that can be found in the following pages summarize the results of our tests. We have only included that part of the results that is common to all the operating systems (see section 4.4.1, *Common tests*, on page 23). The full set of result files, the test programs and the experimental set of classes and systems can all be downloaded from the places indicated in section , *References and sources*, on page 6.

Regina Rexx under OS/2

Test name	
os2.regina	
Test	**Result**
same	Failed
same.rex	Failed
curr	Passed
curr.rex	Passed
path	Passed
path.rex	Passed
lib\samelib	Failed
lib\samelib.rex	Failed
lib\currlib	Passed
lib\currlib.rex	Passed
lib\pathlib	Failed
lib\pathlib.rex	Failed
.\same	Failed
.\same.rex	Failed
.\curr	Passed
.\curr.rex	Passed
.\path	Failed
.\path.rex	Failed
..\dotdotsame	Failed
..\dotdotsame.rex	Failed
..\dotdotcurr	Passed
..\dotdotcurr.rex	Passed
..\dotdotpath	Failed
..\dotdotpath.rex	Failed
lib\..\..\dotdotsame	Failed
lib\..\..\dotdotsame.rex	Failed
lib\..\..\dotdotcurr	Passed
lib\..\..\dotdotcurr.rex	Passed
lib\..\..\dotdotpath	Failed
lib\..\..\dotdotpath.rex	Failed

Object Rexx under OS/2

Test name	
os2.objrexx	
Test	**Result**
same	Failed
same.rex	Failed
curr	Passed
curr.rex	Passed
path	Passed
path.rex	Passed
lib\samelib	Failed
lib\samelib.rex	Failed
lib\currlib	Passed
lib\currlib.rex	Passed
lib\pathlib	Passed
lib\pathlib.rex	Passed
.\same	Failed
.\same.rex	Failed
.\curr	Passed
.\curr.rex	Passed
.\path	Passed
.\path.rex	Passed
..\dotdotsame	Failed
..\dotdotsame.rex	Failed
..\dotdotcurr	Passed
..\dotdotcurr.rex	Passed
..\dotdotpath	Passed
..\dotdotpath.rex	Passed
lib\..\..\dotdotsame	Failed
lib\..\..\dotdotsame.rex	Failed
lib\..\..\dotdotcurr	Passed
lib\..\..\dotdotcurr.rex	Passed
lib\..\..\dotdotpath	Passed
lib\..\..\dotdotpath.rex	Passed

Classic Rexx under OS/2, with the SAA bug fixed

Test name	
os2.rexxsaa.fixed	
Test	**Result**
same	Passed
same.rex	Passed
curr	Passed
curr.rex	Passed
path	Passed
path.rex	Passed
lib\samelib	Passed
lib\samelib.rex	Passed
lib\currlib	Passed
lib\currlib.rex	Passed
lib\pathlib	Passed
lib\pathlib.rex	Passed
.\same	Passed
.\same.rex	Passed
.\curr	Passed
.\curr.rex	Passed
.\path	Passed
.\path.rex	Passed
..\dotdotsame	Passed
..\dotdotsame.rex	Passed
..\dotdotcurr	Passed
..\dotdotcurr.rex	Passed
..\dotdotpath	Passed
..\dotdotpath.rex	Passed
lib\..\..\dotdotsame	Passed
lib\..\..\dotdotsame.rex	Passed
lib\..\..\dotdotcurr	Passed
lib\..\..\dotdotcurr.rex	Passed
lib\..\..\dotdotpath	Passed
lib\..\..\dotdotpath.rex	Passed

Classic Rexx (rexxsaa) under OS/2

Test name	
`os2.rexxsaa`	
Test	**Result**
`same`	Failed
`same.rex`	Failed
`curr`	Failed
`curr.rex`	Passed
`path`	Failed
`path.rex`	Passed
`lib\samelib`	Failed
`lib\samelib.rex`	Failed
`lib\currlib`	Failed
`lib\currlib.rex`	Passed
`lib\pathlib`	Failed
`lib\pathlib.rex`	Failed
`.\same`	Failed
`.\same.rex`	Failed
`.\curr`	Failed
`.\curr.rex`	Passed
`.\path`	Failed
`.\path.rex`	Failed
`..\dotdotsame`	Failed
`..\dotdotsame.rex`	Failed
`..\dotdotcurr`	Failed
`..\dotdotcurr.rex`	Passed
`..\dotdotpath`	Failed
`..\dotdotpath.rex`	Failed
`lib\..\..\dotdotsame`	Failed
`lib\..\..\dotdotsame.rex`	Failed
`lib\..\..\dotdotcurr`	Failed
`lib\..\..\dotdotcurr.rex`	Passed
`lib\..\..\dotdotpath`	Failed
`lib\..\..\dotdotpath.rex`	Failed

Open Object Rexx under Ubuntu

Test name	
`ubuntu.oorexx`	
Test	**Result**
`same`	Passed
`same.rex`	Passed
`curr`	Passed
`curr.rex`	Passed
`path`	Passed
`path.rex`	Passed
`lib/samelib`	Passed
`lib/samelib.rex`	Passed
`lib/currlib`	Passed
`lib/currlib.rex`	Passed
`lib/pathlib`	Passed
`lib/pathlib.rex`	Passed
`./same`	Failed
`./same.rex`	Failed
`./curr`	Passed
`./curr.rex`	Passed
`./path`	Failed
`./path.rex`	Failed
`../dotdotsame`	Failed
`../dotdotsame.rex`	Failed
`../dotdotcurr`	Passed
`../dotdotcurr.rex`	Passed
`../dotdotpath`	Failed
`../dotdotpath.rex`	Failed
`lib/../../dotdotsame`	Passed
`lib/../../dotdotsame.rex`	Passed
`lib/../../dotdotcurr`	Passed
`lib/../../dotdotcurr.rex`	Passed
`lib/../../dotdotpath`	Passed
`lib/../../dotdotpath.rex`	Passed

Regina Rexx under Ubuntu

Test name	
ubuntu.regina	
Test	**Result**
same	Failed
same.rex	Failed
curr	Passed
curr.rex	Passed
path	Passed
path.rex	Passed
lib/samelib	Failed
lib/samelib.rex	Failed
lib/currlib	Passed
lib/currlib.rex	Passed
lib/pathlib	Failed
lib/pathlib.rex	Failed
./same	Failed
./same.rex	Failed
./curr	Passed
./curr.rex	Passed
./path	Failed
./path.rex	Failed
../dotdotsame	Failed
../dotdotsame.rex	Failed
../dotdotcurr	Passed
../dotdotcurr.rex	Passed
../dotdotpath	Failed
../dotdotpath.rex	Failed
lib/../../dotdotsame	Failed
lib/../../dotdotsame.rex	Failed
lib/../../dotdotcurr	Passed
lib/../../dotdotcurr.rex	Passed
lib/../../dotdotpath	Failed
lib/../../dotdotpath.rex	Failed

Open Object Rexx 5.0.0 under Windows

Test name	
`windows.oorexx-5.0.0`	
Test	**Result**
`same`	Passed
`same.rex`	Passed
`curr`	Passed
`curr.rex`	Passed
`path`	Passed
`path.rex`	Passed
`lib\samelib`	Passed
`lib\samelib.rex`	Passed
`lib\currlib`	Passed
`lib\currlib.rex`	Passed
`lib\pathlib`	Passed
`lib\pathlib.rex`	Passed
`.\same`	Failed
`.\same.rex`	Failed
`.\curr`	Passed
`.\curr.rex`	Passed
`.\path`	Failed
`.\path.rex`	Failed
`..\dotdotsame`	Failed
`..\dotdotsame.rex`	Failed
`..\dotdotcurr`	Failed
`..\dotdotcurr.rex`	Passed
`..\dotdotpath`	Failed
`..\dotdotpath.rex`	Failed
`lib\..\..\dotdotsame`	Failed
`lib\..\..\dotdotsame.rex`	Passed
`lib\..\..\dotdotcurr`	Failed
`lib\..\..\dotdotcurr.rex`	Passed
`lib\..\..\dotdotpath`	Failed
`lib\..\..\dotdotpath.rex`	Passed

Open Object Rexx 5.1.0 beta after commit r12651 under Windows

Test name	
`windows.oorexx-5.1.0-beta-r12651`	
Test	**Result**
`same`	Passed
`same.rex`	Passed
`curr`	Passed
`curr.rex`	Passed
`path`	Passed
`path.rex`	Passed
`lib\samelib`	Passed
`lib\samelib.rex`	Passed
`lib\currlib`	Passed
`lib\currlib.rex`	Passed
`lib\pathlib`	Passed
`lib\pathlib.rex`	Passed
`.\same`	Failed
`.\same.rex`	Failed
`.\curr`	Passed
`.\curr.rex`	Passed
`.\path`	Failed
`.\path.rex`	Failed
`..\dotdotsame`	Failed
`..\dotdotsame.rex`	Failed
`..\dotdotcurr`	Passed
`..\dotdotcurr.rex`	Passed
`..\dotdotpath`	Failed
`..\dotdotpath.rex`	Failed
`lib\..\..\dotdotsame`	Passed
`lib\..\..\dotdotsame.rex`	Passed
`lib\..\..\dotdotcurr`	Passed
`lib\..\..\dotdotcurr.rex`	Passed
`lib\..\..\dotdotpath`	Passed
`lib\..\..\dotdotpath.rex`	Passed

Regina Rexx under Windows

Test name	
windows.regina	
Test	**Result**
same	Failed
same.rex	Failed
curr	Passed
curr.rex	Passed
path	Passed
path.rex	Passed
lib\samelib	Failed
lib\samelib.rex	Failed
lib\currlib	Passed
lib\currlib.rex	Passed
lib\pathlib	Failed
lib\pathlib.rex	Failed
.\same	Failed
.\same.rex	Failed
.\curr	Passed
.\curr.rex	Passed
.\path	Failed
.\path.rex	Failed
..\dotdotsame	Failed
..\dotdotsame.rex	Failed
..\dotdotcurr	Passed
..\dotdotcurr.rex	Passed
..\dotdotpath	Failed
..\dotdotpath.rex	Failed
lib\..\..\dotdotsame	Failed
lib\..\..\dotdotsame.rex	Failed
lib\..\..\dotdotcurr	Passed
lib\..\..\dotdotcurr.rex	Passed
lib\..\..\dotdotpath	Failed
lib\..\..\dotdotpath.rex	Failed

The Windows Command Line Interface (`CMD.EXE`)

Test name	
`windows.cmd`	
Test	**Result**
`same`	Failed
`same.rex`	Failed
`curr`	Passed
`curr.rex`	Passed
`pth`	Passed
`pth.rex`	Passed
`lib\samelib`	Failed
`lib\samelib.rex`	Failed
`lib\currlib`	Passed
`lib\currlib.rex`	Passed
`lib\pathlib`	Failed
`lib\pathlib.rex`	Failed
`.\same`	Failed
`.\same.rex`	Failed
`.\curr`	Passed
`.\curr.rex`	Passed
`.\path`	Failed
`.\path.rex`	Failed
`..\dotdotsame`	Failed
`..\dotdotsame.rex`	Failed
`..\dotdotcurr`	Passed
`..\dotdotcurr.rex`	Passed
`..\dotdotpath`	Failed
`..\dotdotpath.rex`	Failed
`lib\..\..\dotdotsame`	Failed
`lib\..\..\dotdotsame.rex`	Failed
`lib\..\..\dotdotcurr`	Passed
`lib\..\..\dotdotcurr.rex`	Passed
`lib\..\..\dotdotpath`	Failed
`lib\..\..\dotdotpath.rex`	Failed

The Windows `SearchPath` API

Test name	
`windows.SearchPath`	
Test	**Result**
`same`	Passed
`same.rex`	Passed
`curr`	Passed
`curr.rex`	Passed
`pth`	Passed
`pth.rex`	Passed
`lib\samelib`	Passed
`lib\samelib.rex`	Passed
`lib\currlib`	Passed
`lib\currlib.rex`	Passed
`lib\pathlib`	Passed
`lib\pathlib.rex`	Passed
`.\same`	Failed
`.\same.rex`	Failed
`.\curr`	Passed
`.\curr.rex`	Passed
`.\path`	Failed
`.\path.rex`	Failed
`..\dotdotsame`	Failed
`..\dotdotsame.rex`	Failed
`..\dotdotcurr`	Passed
`..\dotdotcurr.rex`	Passed
`..\dotdotpath`	Failed
`..\dotdotpath.rex`	Failed
`lib\..\..\dotdotsame`	Passed
`lib\..\..\dotdotsame.rex`	Passed
`lib\..\..\dotdotcurr`	Passed
`lib\..\..\dotdotcurr.rex`	Passed
`lib\..\..\dotdotpath`	Passed
`lib\..\..\dotdotpath.rex`	Passed