ooRexx and Unicode

Updated [JLF May 3, 2010]

Updated [JLF May 4, 2010]

Updated [JLF May 14, 2010]

Updated [JLF May 27, 2010] m17n library (IBM review), PHP, PERL, Parrot

Updated [JLF June 5, 2010] ICU

Updated [JLF June 9, 2010] Falcon

Updated [JLF July 23, 2010] Sandbox investigations

Updated [JLF Sept 14, 2010] MacRuby

Updated [JLF Sept 24, 2010] Review of NetRexx specification

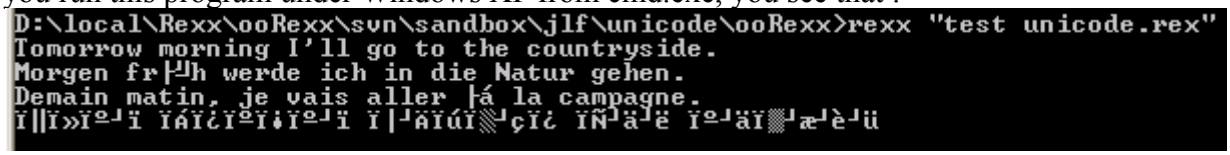
UTF-8 in ooRexx ?

UTF-8 is a multi-byte encoding which can be used as internal format for all the strings.

ooRexx supports UTF-8 in the source files. In fact, it supports probably any multi-bytes encoding but this is a "blind" support, i.e. the interpreter has no knowledge about this encoding, expecting to work on mono-byte strings only.

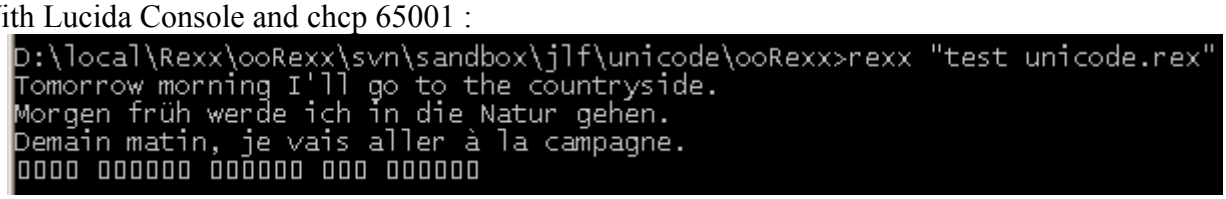
```
sentences = .array~of(-
"Tomorrow morning I'll go to the countryside.",-
"Morgen früh werde ich in die Natur gehen.",-
"Demain matin, je vais aller à la campagne.",-
"غداً صباحاً سَأذهب إلى الرّيف",
)
do s over sentences
    say s
end
```

If you run this program under Windows XP from cmd.exe, you see that :



To display the character correctly in CMD shell, you need to choose the correct code page, e.g., cp65001 for UTF8 : chcp 65001
You also have to choose a font that can display the characters (e.g., Consolas or Lucida Console, NOT Raster font, for Unicode).

With Lucida Console and chcp 65001 :



The arabic characters are not displayed, but it's just because the font doesn't support them. If you copy these garbled characters and past them in you source editor, you will retrieve your original text.

So UTF-8 is supported...But since it's a multi-byte encoding, the current ooRexx String methods will return (sometimes) wrong results (ex : length, charAt). You need specialized methods which understands the encoding rules.

```
s = "aé…"
say s  --> aé…
say s~length --> 6
say s~mapchar('return arg(1)~c2x" "') --> 61 C3 A9 E2 80 A6
```

There is one information missing : the encoding of your source file...

Python and Ruby use a special comment at the begining of the file :

```
# coding=<encoding name>
```

or

```
# -*- coding: <encoding name> -*-
```

or

```
# vim: set fileencoding=<encoding name> :
```

the first or second line must match the regular expression "coding[:=]\s*([-\w.]+)". The first group of this expression is then interpreted as encoding name.

In ooRexx, the encoding could be queried at runtime like that : .context~package~encoding

[JLF May 4, 2010] Any constant string that appears in a source file (package) should be either :

- associated by default with the package's encoding. So any string should have en encoding information which could be retrieved like that : "mystring"~encoding. See the proposition of converting ooDialog to UTF-16 : this string's encoding will be needed to convert to UTF-16 before calling the Windows "W" API.

  This approach is not easy : all the string services should know how to work with any encoding. Ruby 1.9 has taken this approach, to investigate...

- or be converted immediatly to UTF-8. Then we need string services which supports only UTF-8 internally.

Any source which returns some strings (stream, queue, ...) should have a well-know encoding, and any string created by this source should either hold the encoding or be converted immediatly from it to UTF_8.

But is it an ::option encoding="utf-8" ?
or just a comment at the begining of the file ?
Since an ::option can be anywhere in the source file, and repeated several times, it doesn't seems to be a good candidate for that... Remains the comment.
[JLF May 3, 2010] After re-reading, I see that the same remark applies to the comment : can be anywhere in the source file, but must be at first or second line to become a magic comment.

[JLF May 3, 2010] NetRexx supports

```
option utf8
```

If given, clauses following the options instruction are expected to be encoded using UTF-8, so all Unicode characters may be used in the source of the program

```
option noutf8
```

If noutf8 is given, following clauses are assumed to comprise only Unicode characters in the range '\x00' through '\xFF', with the more significant byte of the encoding of each character being 0.

[JLF May 3, 2010] The default encoding in Python is utf-8.

Wide char UnicodeString class in ooRexx ?

This is the Python's approach (in Python 2.x).

The internal ooRexx strings remain byte char strings which contain **encoded** characters.

A UnicodeString instance is created from a String instance by **decoding** the byte chars and converting to UTF-16 or UTF-32. See Wide char strings internally in ooRexx ? for the choice between 16 or 32.

UnicodeString must provide a makeString method which returns an encoded string suitable for current locale's LC_CTYPE.

Pro

Minimal impact.

Cons

For the ooRexx developper, that makes two different implementations to manage.

For the ooRexx programmer, this is a different type from String... Unless we make believe it's a String, like what is done currently for the Integer class...

Wide char strings internally in ooRexx ?

This is the approach described in the experience report (page 6) : replace char by wchar everywhere in the ooRexx sources.

Probably not suitable for the whole interpreter, but could be something to consider for ooDialog. Currently, the "A" Windows API is called, and the conversion occurs there, inside Windows, based on the current locale. If compiling ooDialog with wide chars UTF-16, the "W" API could be called directly, making the dialogs Unicode-enabled. The conversion should be done by ooDialog internally, but the class UnicodeString may help...

[JLF May 3, 2010] ooDialog should remain independant from ICU (unless we include ICU in the delivery). Normally, we just need to use the _tcs family of C runtime functions. ICU services should be needed only for the implementation of the UnicodeString class.

[JLF May 4, 2010] GTK+ uses UTF-8 internally. Most of the Unix-style operating systems use UTF-8 internally. So it seems better to use multi-byte chars in ooRexx instead of wide chars, and to provide string services which supports UTF-8. The case of ooDialog is different : this is a Windows-only sub-system, and for better integration with Windows, it must use UTF-16 chars internally. The conversion to UTF-16 is under the responsability of ooDialog, which lets support code pages that are different from the system's default code page. Typically, we could pass UTF-8 string to ooDialog which would convert it to UTF-16 strings to call the Windows "W" API.

Pro

A unique implementation of String

Under Windows, the GUI can be registered as Unicode GUI.

Cons

Lot of work ! but can be done incrementally because this approach lets build a byte char version

(macro settings).

Needs more memory. But in the current days, increased RAM usage doesn't matter too much.

Only UTF-32 lets support all kind of characters in a single wchar. But even with this encoding, you can have several code points for a single character : Unicode has the concept of combining marks, where the accent would have one point and the letter another. Those are combined into one character when displayed.

With UTF-16, some characters will need two wchars (surrogates). Currently, Windows and Java use UTF-16, so if you want a direct exchange with them, without conversion, you must use UTF-16.

Multi-byte EncodedString in ooRexx ?

This is the Ruby's approach (in Ruby 1.9).

In Ruby 1.8, a String was an array of bytes.

In Ruby 1.9, a String is now some bytes and the rules for interpreting those bytes (each string can have its own encoding). Ruby multilingualization (M17N) of Ruby 1.9 uses the code set independent model (CSI) while many other languages use the Unicode normalization model. To make this original system happen, an encoding convert engine called transcode has been newly added to Ruby 1.9.

UCS vs CSI :

- The UCS normalization model uses a unique character set, which is called Universal Character Set, to handle characters internally. Used by Perl, Python, Java, .NET, Windows, Mac OS X...
- The Code Set Independent (CSI) model does not have a common internal character code set, unlike the UCS Normalization. Under the CSI model, all encodings are handled equally, which means that Unicode is one of character sets. Used by Ruby, Solaris, Citrus...

Pro

[JLF May 14, 2010] Works well for the Japanese community. For a variety of complicated reasons, Japanese encoding, such as SHIFT-JIS, are not considered to losslessly encode into UTF-8. As a result, Ruby has a policy of not attempting to simply encode any inbound String into UTF-8.

Cons

More complex and slower than wide char.

Notes about ICU

Updated [JLF June 5, 2010]

http://userguide.icu-project.org

Current version at writing this section is 4.4.1

UText

Seems to be the interface needed by ooRexx, assuming we use UTF-8.

UText: Added in ICU4C 3.4 as a technology preview. Intended to be the strategic text access API for use with ICU. C API, high performance, writable, supports native indexes for efficient non-UTF-16 text storage. It allows for high-performance operation through the use of storage-native indexes (for efficient use of non-UTF-16 text) and through accessing multiple characters per function call. Code point iteration is available with functions as well as with C macros, for maximum performance. UText is also writable, mostly patterned after Replaceable.

ICU uses signed 32-bit integers (int32_t) for lengths and offsets. Because of internal computations, strings (and arrays in general) are limited to 1G base units or 2G bytes, whichever is smaller.

Strings are either terminated with a NUL character (code point 0, U+0000) or their length is specified. In the latter case, it is possible to have one or more NUL characters inside the string.

Locale

From a geographic perspective, a locale is a place. From a software perspective, a locale is an ID used to select information associated with a a language and/or a place. ICU locale information includes the name and identifier of the spoken language, sorting and collating requirements, currency usage, numeric display preferences, and text direction (left-to-right or right-to-left, horizontal or vertical).

The ICU services support all major locales with language and sub-language pairs. The sub-language generally corresponds to a country. One way to think of this is in terms of the phrase "X language as spoken in Y country." The way people speak or write a particular language might not change dramatically from one country to the next (for example, German is spoken in Austria, Germany, and Switzerland). However, cultural conventions and national standards often differ a great deal.

A locale ID specifies a language and region enabling the software to support culturally and linguistically appropriate information for each user. A locale object represents a specific geographical, political, or cultural region. As a programmatic expression of locale IDs, ICU provides the C++ locale class. In C, Application Programming Interfaces (APIs) use simple C strings for locale IDs.

The following links provide additional useful information regarding ISO standards: ISO-639 , and an ISO Country Code, ISO-3166 . For example, Italian, Italy, and Euro are designated as: it_IT_EURO.

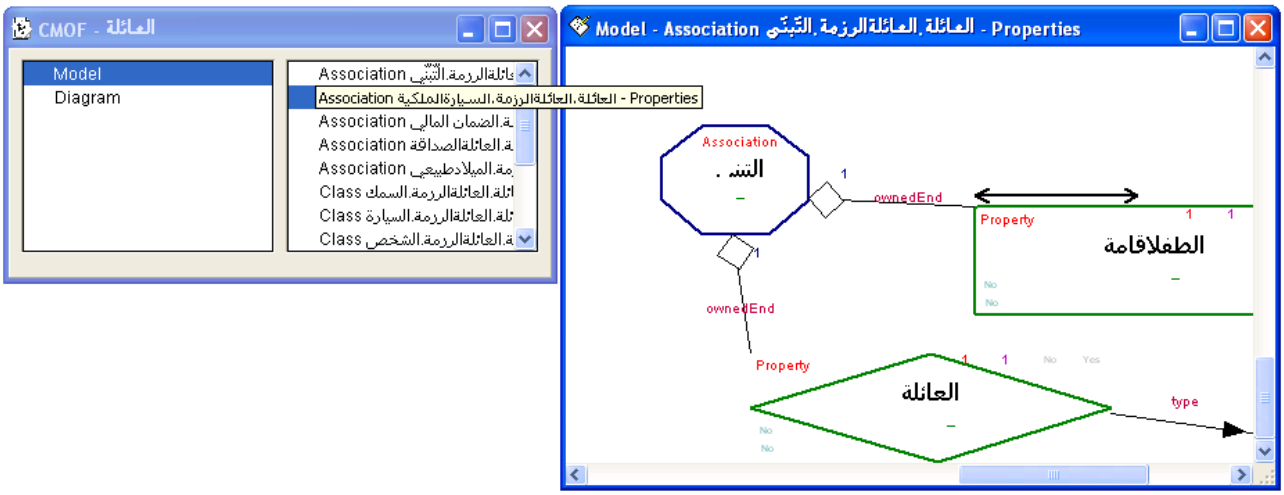
Darwin

http://icu.darwinports.com/

Experience report

The directory "example of migration" contains a selection of notes taken during a past project to migrate a graph editor to Unicode. Some selections of source files are also available (stripped versions), to illustrate the changes.

The goal was to use UTF-16 strings internally (wchar*), instead of ANSI strings (char*), to use the TCHAR abstraction and to create windows that are registered as Unicode windows. It was then possible to display any Unicode string in the menus and in the graphs.



The target platform was Windows only.

There was no need of a library like ICU because the main goal was to support Unicode strings in the GUI, not to analyze/transform the Unicode strings.

Read first the release notes for an overview of the changes visible to the API user.

See the file Encodings - Unicode - i18n.odt for more detailled notes. The chapters in direct relation with the migration process are :

- Unicode Programming Summary
- Using Generic-Text Mappings
- String Manipulation
- Files and Stream
- Windows & Unicode

The migration process was quite mechanical :

- replace char by gtchar everywhere.
- use proper C run-time functions for Unicode string handling. See the file migration notes.txt, there is a table of functions showing the changes made internally. Ex : isalnum has been replaced by _istalnum.
- compile often (in wide char mode), let the compiler tell you what's wrong, fix...
- sometimes you have to adapt the code, in particular at the boundaries of the API or when reading/writing streams. Here you have to manage the conversion between byte and wide chars.
- For the API, you have to decide if you want to expose a dual API "A" and "W", similar to

what Windows offers. For binary compatibility, you should export "myFunction" and "myFunctionW". In the source files given as examples, we have used the naming convention "myFunctionA" and "myFunctionW".

At the end, you can generate two versions of executable :

- a version which uses byte chars internally, not Unicode enabled but 100% compatible with your previous version (assuming that you dont't use the "A" naming convention)
- a version which uses wide chars internally, Unicode enabled. The legacy applications are client of the byte char API. To take advantage of the wide char API, they must define the macro TOOL_WIDE_API and recompile their sources (after adaptation to support wide char strings).

Links

"A" and "W" API

Unicode and Non-Unicode ODBC Drivers

http://www.datadirect.com/developer/odbc/unicode/odbc-driver/index.ssp

Falcon

[JLF June 9, 2010]

http://falconpl.org/index.ftd?page_id=sitewiki&prj_id=_falcon_site&sid=wiki&pwid=Survival%20Guide&wid=Survival%3ABasic+Structures

International strings

Falcon strings can contain any Unicode character. The Falcon compiler can input source files written in various encodings. UTF-8 and UTF-16 and ISO8859-1 (also known as Latin-1) are the most common; Unicode characters can also be inserted directly into a string via escapes.

When assigning an integer number between 0 and 2^32 (that is, the maximum allowed by the Unicode standard) to a string portion via the array accessor operator (square brackets), the given portion will be changed into the specified Unicode character.

```
1.string = "Beta: "
2.string[5] = 0x3B2
3.printl( string )  // will print Beta:β
```

Accessing the nth character with the square brackets operator will cause a single character string to be produced. However, it is possible to query the Unicode value of the nth character with the bracket-star operator using the star square operator ([*]):

```
1.string = "Beta:β"
2.i = 0
3.while i < string.len()
4.    > string[i], "=", string[* i++]
5.end
```

This code will print each character in the string along with its Unicode ID in decimal format.

String polymorphism

In Falcon, to store and handle efficiently strings, strings are built on a buffer in which each character occupies a fixed space. The size of each character is determined by the size in bytes needed by the widest character to be stored. For Latin letters, and for all the Unicode characters whose code is less than 256, only one byte is needed. For the vast majority of currently used alphabets, including Chinese, Japanese, Arabic, Hebrew, Hindi and so on, two bytes are required. For unusual symbols like musical notation characters four bytes are needed. In this example:

```
1.string = "Beta: "
2.string[5] = 0x3B2
3.printl( string )  // will print "Beta:β"
```

the string variable was initially holding a string in which each character could have been

represented with one byte.

The string was occupying exactly six bytes in memory. When we added β the character size requirement changed. The string has been copied into a wider space. Now, twelve characters are needed as β Unicode value is 946 and two bytes are needed to represent it.

When reading raw data from a file or a stream (i.e. a network stream), the incoming data is always stored byte per byte in a Falcon string. In this way binary files can be manipulated efficiently; the string can be seen just as a vector of bytes as using the [*] operator gives access to the nth byte value. This allows for extremely efficient binary data manipulation.

However, those strings are not special. They are just loaded by inserting 0-255 character values into each memory slot, which is declared to be 1 byte long. Inserting a character requiring more space will create a copy of each byte in the string in a wider memory area.

Files and streams can be directly loaded using transcoders. With transcoder usage, loaded strings may contain any character the transcoder is able to recognize and decode.

Strings can be saved to files by both just considering their binary content or by filtering them through a transcoder. In the case that a transcoded stream is used, the output file will be a binary file representing the characters held in the string as per the encoding rules.

Although this mixed string valence, that uses fully internationalized multi-byte character sequences and binary byte buffers, could be confusing at first, it allows for flexible and extremely efficient manipulation of binary data and string characters depending on the need.

It is possible to know the number of bytes occupied by every character in a string through the String.charSize method of each string; the same method allows to change the character size at any moment. See the following example:

```
1.str = "greek: αβγ"
2.> str.charSize()      // prints 2
3.str.charSize( 1 )      // squeeze the characters
4.> str                       // "greek: " + some garbage
```

This may be useful to prepare a string to receive international characters at a moment's notice, avoiding paying the cost for character size conversion. For example, suppose you're reading a text file in which you expect to find some international characters at some point. By configuring the size of the accumulator string ahead of time you prevent the overhead of determining character byte size giving you a constant insertion time for each operation:

```
1.str = ""
2.str.charSize( 2 )
3.file = ...
4.
5.while not file.eof()
6.   str += file.read( 512 )
7.end
```

String usage in C/C++ programs

The `AutoCString` class is a simple and efficient way to transform a Falcon string in an UTF-8 encoded C string. It uses a stack area if the string is small enough, and eventually accounts for proper allocation otherwise. Falcon::String has a toCString() member that can fill a C string with an UTF-8 representation of the internal string data.

```
1.catch( Falcon::Error* err )
2.{
```

```
3.   Falcon::AutoCString edesc( err->toString() );
4.   std::cerr << edesc.c_str() << std::endl;
5.   err->decref();
6.   return 1;
7.}
```

On wide-character environments (Visual Studio, wchar_t* enabled STL with gcc and so on), you'll prefer `AutoWString`, which transform a falcon string in a system specific wchar_t* string. Falcon::String has a toWideString member that can fill a C string with an wide-char representation of the internal string data.

Finally, a fast debug `printf()` may use the `String::c_ize()` method. That method just adds an extra 0 at the end of the internal buffer, in appropriate character width, so that a printf can be performed directly on the inner data (which is available through the `String::getRawStorage()` method).

GTK

[JLF May 4, 2010]

Uses UTF-8 internally.

Unicode Manipulation

http://www.gtk.org/api/2.6/glib/glib-Unicode-Manipulation.html

Windows portability for GNOME software

http://tml.pp.fi/fosdem-2006.pdf

Filename character set :

- File system uses Unicode (UTF-16)
- Each machine has a fixed ”system codepage”: a single- or variable-length (double-byte) character set
- Single-byte codepages: CP1252 etc. For European, Middle East languages, Thai, etc
- Double-byte codepages: In East Asia
- It's quite possible to have file names on a machine that can't be represented in the system codepage. Occurs in East Asia, and for Western Europeans who exchange documents with Greece, Russia, etc
- All file name APIs in the C library have two versions:
    - normal one (fopen) uses system codepage,
    - the wide character one (_wfopen) uses wchar_t
- But, forget all the above, just use UTF-8 and GLib
- GLib and GTK+ APIs use UTF-8
- gstdio wrappers for UTF-8 pathnames: g_open(), g_fopen(), g_dir_*(), g_stat() etc
- Other libraries like libxml2 and gettext don't expect UTF-8 pathnames
- Need to pass them system codepage filenames

- g_win32_locale_filename_from_utf8() should work in most cases for existing files.

Haskell

Bindings to the ICU library

http://www.serpentine.com/bos/files/haddock/text-icu/

http://hackage.haskell.org/package/text-icu

unicode-normalization: Unicode normalization using the ICU library

http://hackage.haskell.org/package/unicode-normalization

uconv: String encoding conversion with ICU (experimental)

http://hackage.haskell.org/package/uconv

text: An efficient packed Unicode text type.

http://hackage.haskell.org/package/text

An efficient packed, immutable Unicode text type (both strict and lazy), with a powerful loop fusion optimization framework.

The Text type represents Unicode character strings, in a time and space-efficient manner. This package provides text processing capabilities that are optimized for performance critical use, both in terms of large data quantities and high speed.

The Text type provides character-encoding, type-safe case conversion via whole-string case conversion functions. It also provides a range of functions for converting Text values to and from ByteStrings, using several standard encodings (see the text-icu package for a much larger variety of encoding functions).

Efficient locale-sensitive support for text IO is also supported.

deunicode: Get rid of unicode (utf-8) symbols in Haskell sources

A very simple-minded program to replace utf-8 encoded unicode operators in Haskell source files with their equivalent in ascii. It takes no arguments and acts as a pure filter from stdin to stdout.

http://hackage.haskell.org/package/deunicode

utf8-string: Support for reading and writing UTF8 Strings

A UTF8 layer for IO and Strings. The utf8-string package provides operations for encoding UTF8 strings to Word8 lists and back, and for reading and writing UTF8 without truncation.

http://hackage.haskell.org/package/utf8-string

hxt-unicode: Unicode en-/decoding functions for utf8, iso-latin-* and other encodings

Unicode encoding and decoding functions for utf8, iso-latin-* and somes other encodings, used in the Haskell XML Toolbox. ISO Latin 1 - 16, utf8, utf16, ASCII are supported. Decoding is done with lasy functions, errors may be detected or ignored.

http://hackage.haskell.org/package/hxt-unicode

regex-tdfa-utf8: This combines regex-tdfa with utf8-string to allow searching over UTF8 encoded lazy bytestrings.

http://hackage.haskell.org/package/regex-tdfa-utf8

IBM

[JLF May 27, 2010]

m17n Multilingualization library

http://www.ibm.com/developerworks/linux/library/l-m17n/

Unicode surrogate programming with the Java language

http://www.ibm.com/developerworks/java/library/j-unicode/index.html

Java

http://jelmer.jteam.nl/2007/08/12/on-character-set-encodings/

MacRuby

http://www.macruby.org/

The primitive Ruby classes (String, Array, and Hash) have been re-implemented on top of their Cocoa equivalents (respectively, NSString, NSArray, and NSDictionary).

As an example, String is no longer a class, but a pointer (alias) to NSMutableString. All strings in MacRuby are genuine Cocoa strings and can be passed (without conversion) to underlying C or Objective-C APIs that expect Cocoa strings.

The whole String interface was re-implemented on top of NSString. This means that you can call any method of String on any Cocoa string. Because Cocoa strings can be either mutable and immutable, if you try to call a method that is supposed to modify its receiver on an immutable string, a runtime exception will be raised.

Because NSString was not designed to handle bytestrings, MacRuby will automatically (and silently) create an NSData object when necessary, attach it to the string object, and proxy the methods to its content. This will typically be used when you read binary data from a file or a network socket.

http://www.infoq.com/MacRuby

MacRuby 0.6 :

The String class has been changed. It is now a fresh new implementation that can handle both character and byte strings. It also uses the ICU framework to perform encoding conversions on the

fly. This new class inherits from NSMutableString. Symbol was also rewritten to handle multibyte (Unicode) characters.

Finally, the Regexp class has been totally rewritten in this release. It is now using the ICU framework instead of Oniguruma for regular expression compilation and pattern matching. Since ICU is thread-safe, MacRuby 0.6 allows multiple threads to utilize regular expressions in a very efficient way, which was not possible previously.

http://macruby.labs.oreilly.com/

MacRuby is Apple's implementation of the Ruby programming Language. More precisely, it is a Ruby implementation that uses the well known and proven Objective-C runtime giving you direct native access to all the OS X libraries. The end result is a first-class, compilable scripting language designed to develop applications for the OS X platform.

http://lists.macosforge.org/pipermail/macruby-devel/2010-August/005762.html

MacRuby doesn't really honor the script encoding setting. It uses UTF-8 by default for pretty much everything. I suspect it's not going to change any time soon, though better compatibility could be added once we approach 1.0.

http://lists.macosforge.org/pipermail/macruby-devel/2009-April/001552.html

Strings, Encodings and IO

Microsoft

Code page identifiers

http://msdn.microsoft.com/en-us/library/dd317756

Parrot

[JLF May 27, 2010]

Essay

What the heck is: A string

http://www.sidhe.org/~dan/blog/archives/000255.html

Strings, some practical advice

http://www.sidhe.org/~dan/blog/archives/cat_parrot_notes.html#000256

The Pain of Text

http://www.sidhe.org/~dan/blog/archives/cat_parrot_notes.html#000294

Parrot strings & Grapheme Cluster

http://docs.parrot.org/parrot/devel/html/docs/pdds/pdd28_strings.pod.html

Parrot was designed from the outset to support multiple string formats: multiple character sets and multiple encodings. We don't standardize on Unicode internally, converting all strings to Unicode strings, because for the majority of use cases it's still far more efficient to deal with whatever input data the user sends us.

Normalization Form G (NFG). In NFG, every grapheme is guaranteed to be represented by a single codepoint. Graphemes that don't have a single codepoint representation in Unicode are given a dynamically generated codepoint unique to the NFG string.

An NFG string is a sequence of signed 32-bit Unicode codepoints. It's equivalent to UCS-4 except for the normalization form semantics. UCS-4 specifies an encoding for Unicode codepoints from 0 to 0x7FFFFFFF. In other words, any codepoints with the first bit set are undefined. NFG interprets the unused bit as a sign bit, and reserves all negative codepoints as dynamic codepoints. A negative codepoint acts as an index into a lookup table, which maps between a dynamic codepoint and its associated decomposition.

Serbo-Croat is sometimes written with Cyrillic letters rather than Latin letters. Unicode doesn't have a single composed character for the Cyrillic equivalent of the Serbo-Croat LATIN SMALL LETTER I WITH DOUBLE GRAVE, so it is represented as a decomposed pair CYRILLIC SMALL LETTER I (0x438) with COMBINING DOUBLE GRAVE ACCENT (0x30F). When our Russified Serbo-Croat string is converted to NFG, it is normalized to a single character having the codepoint 0xFFFFFFFFF (in other words, -1 in 2's complement). At the same time, Parrot inserts an entry into the string's grapheme table at array index -1, containing the Unicode decomposition of the grapheme 0x00000438 0x000000030F.

Parrot's internal strings (STRINGs) have the following structure:

```
struct parrot_string_t {

    Parrot_UInt flags; // Binary flags used for garbage collection, copy-on-write tracking, and
other metadata

    void *     _bufstart; // pointer to the buffer for the string data

    size_t     _buflen; // size of the buffer in bytes

    UINTVAL     bufused; // amount of the buffer currently in use, in bytes

    UINTVAL     strlen; // length of the string, in bytes

    UINTVAL     hashval; // cache of the hash value of the string, for rapid lookups when the
string is used as a hash key

    const struct _encoding *encoding; // How the data is encoded (e.g. fixed 8-bit characters, UTF-
8, or UTF-32). The encoding structure specifies the encoding (by index number and by name, for ease
of lookup), the maximum number of bytes that a single character will occupy in that encoding, as
well as functions for manipulating strings with that encoding.

    const struct _charset  *charset; // What sort of string data is in the buffer, for example
ASCII, EBCDIC, or Unicode.The charset structure specifies the character set (by index number and by
name) and provides functions for transcoding to and from that character set.

};
```

What is NFG and why you want Parrot to have it.

http://www.parrot.org/content/what-nfg-why-you-want-parrot-have-it

Encodings, charsets and how NFG fits in there.

http://www.parrot.org/content/encodings-charsets-and-how-nfg-fits-there

The 'charset' deals with the set of characters that is used in the string, it can be one of ASCII, ISO-

8859-1, Unicode or binary (technically, binary isn't really a charset but rather the absence of one, it means "I'm just a stream of bytes, no characters here). In Unicode terms it is a character repertoire, a collection of characters, a well-defined subset of the Unicode character space.

It is the charset's job to know everything there is to know about the characters that make up the string: composition, decomposition, case changes, character classes and whether a given codepoint is a member of the character set. It also knows how to convert strings to and from other charsets, but charsets are all blissfully ignorant on one small detail: The representation of characters inside the string.

That's the part that the 'encoding' is expected to handle. The encodings available to parrot today are: fixed_8, utf8, ucs2 and utf16.

UCS-4, NFG and how the grapheme table makes it awesome.

http://www.parrot.org/content/ucs-4-nfg-and-how-grapheme-tables-makes-it-awesome

UCS-4 was defined in the original ISO 10646 standard as a 31-bit encoding form in which each encoded character in the Universal Character Set (That's the UCS in the name, for those wondering.) is represented by a 32-bit integers between 0x00000000. and 0x7FFFFFFF. Tremendously wasteful of your memory, but still useful since now you can fit any weird character you can think of in one 'position' and you can go back to doing O(1) random access into your strings. Mostly.

On paper UCS-4 looks like it was meant to solve all of our problems, but didn't. It solves a few of the problems, like fixed-with encoding of codepoints, but it still leaves some unfinished business. The biggest issue is combining characters.

Dynamic code points, the grapheme tables and not getting your services denied.

http://www.parrot.org/content/dynamic-code-points-grapheme-tables-and-not-getting-your-services-denied

When converting a string into NFG we will dynamically create new codepoints for sequences of composing characters that do not map to a single Unicode code point. The information needed to turn that new codepoint back into a stream of valid unicode codepoints is stored in the grapheme table. With one entry per created code point, and the relative rarity of graphemes that lack an Unicode code point, this table is expected to be quite small most of the time, you would have to be using some rather odd inputs for it to grow beyond a hundred entries.

string manipulation based on graphemes

http://www.mail-archive.com/perl-unicode@perl.org/msg01578.html

http://search.cpan.org/~sadahiro/

http://www.nntp.perl.org/group/perl.perl6.language/2009/05/msg31560.html

Parrot uses ICU, if installed.

[JLF Sept 24, 2010] Removed the source excerpts.

src/ops/core_ops.c

src/string/api.c

This file implements the non-ICU parts of the Parrot string subsystem.

src/string/primitives.c

This file collects together all the functions that call into the ICU

src/string/charset/unicode.c

This file implements the charset functions for unicode data

src/string/charset/ucs2.c

src/string/charset/ucs4.c

src/string/charset/utf16.c

UTF-16 encoding with the help of the ICU library.

src/string/charset/utf8.c

Does NOT depend on ICU...

PERL

[JLF May 27, 2010]

http://perldoc.perl.org/perlunicode.html

PHP

[JLF May 27, 2010]

Mar 11 2010 : PHP 6 trunk moved to a branch because of problems with Unicode

http://news.php.net/php.internals/47120

http://schlueters.de/blog/archives/128-Future-of-PHP-6.html

ICU and PHP

http://devzone.zend.com/article/4799-Internationalization-in-PHP-5.3

Human Language and Character Encoding Support

http://fr2.php.net/manual/en/refs.international.php

Python

Python 2.x contains Unicode support. It has a new fundamental data type `unicode', representing a

Unicode string, a module `unicodedata' for the character properties, and a set of converters for the most important encodings.

Python 3.x uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. All text is Unicode; however encoded Unicode is represented as binary data. The type used to hold text is str, the type used to hold data is bytes. The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.x raises TypeError, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but you would get UnicodeDecodeError if it contained non-ASCII values.

In file include/unicodeobject.h (Python 3.X) :

```
/* Setting Py_UNICODE_WIDE enables UCS-4 storage.  Otherwise, Unicode
   strings are stored as UCS-2 (with limited support for UTF-16) */

#if Py_UNICODE_SIZE >= 4
#define Py_UNICODE_WIDE
#endif

/* Windows has a usable wchar_t type (unless we're using UCS-4) */
# if defined(MS_WIN32) && Py_UNICODE_SIZE == 2
#  define HAVE_USABLE_WCHAR_T
#  define PY_UNICODE_TYPE wchar_t
# endif

# if defined(Py_UNICODE_WIDE)
#  define PY_UNICODE_TYPE Py_UCS4
# endif

/* Py_UNICODE is the native Unicode storage format (code unit) used by
   Python and represents a single Unicode element in the Unicode
   type. */

typedef PY_UNICODE_TYPE Py_UNICODE;

typedef struct {
    PyObject_HEAD
    Py_ssize_t length;              /* Length of raw Unicode data in buffer */
    Py_UNICODE *str;         /* Raw Unicode buffer */
    long hash;               /* Hash value; -1 if not set */
    PyObject *defenc;        /* (Default) Encoded version as Python
                                string, or NULL; this is used for
                                implementing the buffer protocol */
} PyUnicodeObject;
```

Defining Python source encoding

http://www.python.org/dev/peps/pep-0263/

Python Unicode Objects
http://effbot.org/zone/unicode-objects.htm

The Truth About Unicode In Python
http://www.cmlenz.net/archives/2008/07/the-truth-about-unicode-in-python

All About Python and Unicode
http://boodebr.org/main/python/all-about-python-and-unicode

Encodings in Python
http://www.umiacs.umd.edu/~aelkiss/xml/python/encode4.html

How to Use UTF-8 with Python
http://evanjones.ca/python-utf8.html

ICU wrapping
http://pypi.python.org/pypi/PyICU

Codec registry and base classes
http://docs.python.org/library/codecs.html

Ruby

Understanding M17N
http://blog.grayproductions.net/articles/understanding_m17n

The design and implementation of Ruby M17N
http://yokolet.blogspot.com/2009/07/design-and-implementation-of-ruby-m17n.html

Consider the ICU Library for Improving and Expanding Unicode Support
http://redmine.ruby-lang.org/issues/show/2034

[JLF May 14, 2010]
Oniguruma, a regular expression library which is standard with Ruby 1.9
http://oniguruma.rubyforge.org/
http://redmine.ruby-lang.org/issues/show/1889

Ropes: an Alternative to Strings (1995) - Cited by Yui NARUSE

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.9450

From Yui NARUSE :

Rope is:

* fast string concatnation

* fast substring get

* can't change substring

* slow index access to a character

But Ruby's string is mutable.

This seems a critical issue for rope.

Moreover Ruby users often use regexp match to strings.

I don't think rope has enough merit to implement despite such tough environment

[JLF May 14, 2010] Breaks compatibility for legacy applications. This section is a summary of

http://yehudakatz.com/2010/05/05/ruby-1-9-encodings-a-primer-and-the-solution-for-rails/

Because it’s extremely tricky for Ruby to be sure that it can make a lossless conversion from one encoding to another (Ruby supports almost 100 different encodings), the Ruby core team has decided to raise an exception if two Strings in different encodings are concatenated together.

There is one exception to this rule. If the bytes in one of the two Strings are all under 127 (and therefore valid characters in ASCII-7), and both encodings are compatible with ASCII-7 (meaning that the bytes of ASCII-7 represent exactly the same characters in the other encoding), Ruby will make the conversion without complaining.

By default, Strings with no encoding in Ruby are tagged with the ASCII-8BIT encoding, which is an alias for BINARY. Essentially, this is an encoding that simply means “raw bytes here”. Almost all of the encoding problems reported by users in the Rails bug tracker involved ASCII-8BIT Strings. Two reasons :

- Database drivers generally didn’t properly tag Strings they retrieved from the database with the proper encoding. This involves a manual mapping from the database’s encoding names to Ruby’s encoding names. As a result, it was extremely common from database drivers to return Strings with characters outside of the ASCII-7 range (because the original content was encoded in the database as UTF-8 or ISO-8859-1/Latin-1)
- There is a second large source of BINARY Strings in Ruby. Specifically, data received from the web in the form of URL encoded POST bodies often do not specify the content-type of the content sent from forms. In many cases, browsers send POST bodies in the encoding of the original document, but not always. In addition, some browsers say that they’re sending content as ISO-8859-1 but actually send it in Windows-1251.

Solution proposed by Yehuda Katz :

By default, Ruby should continue to support Strings of many different encodings, and raise exceptions liberally when a developer attempts to concatenate Strings of different encodings. This would satisfy those with encoding concerns that require manual resolution.

Additionally, you would be able to set a preferred encoding. This would inform drivers at the boundary (such as database drivers) that you would like them to convert any Strings that they tag with an encoding to your preferred encoding immediately. By default, Rails would set this to UTF-

8, so Strings that you get back from the database or other external source would always be in UTF-8.

If a String at the boundary could not be converted (for instance, if you set ISO-8859-1 as the preferred encoding, this would happen a lot), you would get an exception as soon as that String entered the system.

In practice, almost all usage of this setting would be to specify UTF-8 as a preferred encoding. From your perspective, if you were dealing in UTF-8, ISO-8859-* and ASCII (most Western developers), you would never have to care about encodings.

Even better, Ruby already has a mechanism that is mostly designed for this purpose. In Ruby 1.9, setting Encoding.default_internal tells Ruby to encode all Strings crossing the barrier via its IO system into that preferred encoding. All we'd need, then, is for maintainers of database drivers to honor this convention as well.

[JLF May 15, 2010]

Encodings, Unabridged

http://yehudakatz.com/2010/05/17/encodings-unabridged/

Character encoding auto-detection

http://github.com/speedmax/rchardet/tree

Unicode

Regular expressions
[JLF May 14, 2010]

Unicode Technical Standard #18

http://unicode.org/reports/tr18/

Unicode text segmentation

http://www.unicode.org/reports/tr29

Grapheme clusters

Regular expressions

Unix/Linux

The Unicode HOWTO

http://www.tldp.org/HOWTO/Unicode-HOWTO.html

UTF-8 and Unicode FAQ for Unix/Linux

http://www.cl.cam.ac.uk/~mgk25/unicode.html

m17n Library

http://www.m17n.org/m17n-lib-en/index.html

W3C

Migrating to Unicode

http://www.w3.org/International/articles/unicode-migration/

Wide char in C

http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap06.html#tag_06_03

Tchar I18N Text Abstraction

http://www.ioplex.com/~miallen/libmba/dl/docs/ref/libmba.html

Unsorted

NRSI: Computers & Writing Systems

http://scripts.sil.org

State of Text Rendering

http://behdad.org/text/

This document describes the basic ideas of I18N; it's written for programmers and package maintainers of Debian GNU/Linux and other UNIX-like platforms. The aim of this document is to offer an introduction to the basic concepts, character codes, and points where care should be taken when one writes an I18N-ed software or an I18N patch for an existing software.

http://www.debian.org/doc/manuals/intro-i18n/

NetRexx

Links

[JLF May 3, 2010]

VM/ESA Network Computing with Java and NetRexx

http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg245148.html

You can use the System class to find the default codepage set by the Java virtual machine.

```
cp = System.getProperty('file.encoding' )
Say 'we use codepage' cp
```

Translating between EBCDIC and ASCII : The InputStreamReader and OutputStreamWriter classes can be used for sending/receiving ASCII :

```
connection = Socket(host,port)
asciiIn = InputStreamReader( connection.getInputStream() ,'8859_1')
asciiOut= OutputStreamWriter(connection.getOutputStream(),'8859_1')
fromNet = BufferedReader(asciiIn)
```

Java's character stream classes define a method which allows you to find out which codepage has been defined (or defaulted) for the stream:

```
say 'Encoding of "instream"  is' inStream.GetEncoding()
```

Review of specification 1.00

[JLF Sept 24, 2010]

From Rick

The biggest problems with trying to do this without breaking some features of the language that explicitly assume that a character is an 8-bit byte with the range '00'x-'ff'x. For example, xrange(), c2x(), x2c(), c2d(), d2c(). Hex and binary literals also present some interesting challenges. There are also lots of places in the APIs where strings are passed around using similar assumptions, so these transitions need to be defined. Also versions of the APIs that can deal with UNICode strings will need to be defined. Re-implementing the base string methods should be straightforward, but without first focusing on the issues where the base assumptions of the language might have to change, I'm not sure much is gained from that experiment.

...

My initial thought was to follow the lead blazed by NetRexx, but Mike pretty much punted on the issues as well. The I/O model is still largely the Java one, which has a lot of support in it to deal with encoding issues. Things like c2x(), etc. only work on single characters rather than strings, which basically punts on that issue. And the language literal strings were defined from the outset with unicode in mind, so that compatibility issue didn't exist either. And there are no native APIs in netrexx, so that was no help.

nlrdef p3

Character Sets

Programming in the NetRexx language can be considered to involve the use of two character sets. The first is used for expressing the NetRexx program itself, and is the relatively small set of characters described in the next section. The second character set is the set of characters that can be used as character data by a particular implementation of a NetRexx language processor. This character set may be limited in size (sometimes to a limit of 256 different characters, which have a convenient 8-bit representation), or it may be much larger. The Unicode[4] character set, for example, allows for 65536 characters, each encoded in 16 bits.

Usually, most or all of the characters in the second (data) character set are also allowed within a NetRexx program, but only within commentary or immediate (literal) data.

The NetRexx language explicitly defines the first character set, in order that programs will be portable and understandable;

nlrdef p6

Within literal strings, characters that cannot safely or easily be represented (for example "control characters") may be introduced using an escape sequence. An escape sequence starts with a backslash ("\"), which must then be followed immediately by one of the following (letters may be in either uppercase or lowercase):

`xhh` the escape sequence represents a character whose encoding is given by the two hexadecimal digits ("`hh`") following the "`x`". If the character encoding for the implementation requires more than two hexadecimal digits, they are padded with zero digits on the left.

`uhhhh` the escape sequence represents a character whose encoding is given by the four hexadecimal digits ("`hhhh`") following the "`u`". It is an error to use this escape if the character encoding for the implementation requires fewer than four hexadecimal digits.

Examples:

```
'You shouldn\'t' /* Same as "You shouldn't" */
'\x6d\u0066\x63' /* In Unicode: 'mfc' */
'\\\u005C' /* In Unicode, two backslashes */
```

nlrdef p73

utf8

If given, clauses following the **options** *instruction are expected to be encoded using UTF-8, so all Unicode characters may be used in the source of the program.*

In UTF-8 encoding, Unicode characters less than `'\u0080'` *are represented using one byte (whose most-significant bit is 0), characters in the range* `'\u0080'` *through* `'\u07FF'` *are encoded as two bytes, in the sequence of bits:*

`110xxxxx 10xxxxxx`

where the eleven digits shown as `x` *are the least significant eleven bits of the character, and characters in the range* `'\u0800'` *through* `'\uFFFF'` *are encoded as three bytes, in the sequence of bits:*

`1110xxxx 10xxxxxx 10xxxxxx`

where the sixteen digits shown as `x` *are the sixteen bits of the character.*

If **noutf8** *is given, following clauses are assumed to comprise only Unicode characters in the range* `'\x00'` *through* `'\xFF'`*, with the more significant byte of the encoding of each character being 0.*

nlrdef p120

9. Conversion between character encodings and decimal or hexadecimal is dependent on the machine representation (encoding) of characters and hence will return appropriately different results for Unicode, ASCII, EBCDIC, and other implementations.

nlrdef p120

c2d()
Coded character to decimal. Converts the encoding of the character in *string* (which must be exactly one character) to its decimal representation. The returned string will be a non-negative number that represents the encoding of the character and will not include any sign, blanks, insignificant leading zeros, or decimal part.
Examples:
```
'M'.c2d == '77' -- ASCII or Unicode
'7'.c2d == '247' -- EBCDIC
'\r'.c2d == '13' -- ASCII or Unicode
'\0'.c2d == '0'
```
The `c2x` method (see page 124) can be used to convert the encoding of a character to a hexadecimal representation.
JLF : in ooRexx, the argument can be a string of any length. In NetRexx, the string is limited to one character. I don't see where is the problem. For me, this function iterates over the bytes of the string.

c2x()
Coded character to hexadecimal. Converts the encoding of the character in *string* (which must be exactly one character) to its hexadecimal representation (unpacks). The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. Insignificant leading zeros are removed.
Examples:
```
'M'.c2x == '4D' -- ASCII or Unicode
'7'.c2x == 'F7' -- EBCDIC
'\r'.c2x == 'D' -- ASCII or Unicode
'\0'.c2x == '0'
```
The `c2d` method (see page 124) can be used to convert the encoding of a character to a decimal number.
JLF : in ooRexx, the argument can be a string of any length. In NetRexx, the string is limited to one character. I don't see where is the problem. For me, this function iterates over the bytes of the string.

nlrdef p125

datatype(option)

`A` (Alphanumeric); returns 1 if *string* only contains characters from the ranges “a-z”, “A-Z”, and “0-9”.
`L` (Lowercase); returns 1 if *string* only contains characters from the range “a-z”.
`M` (Mixed case); returns 1 if *string* only contains characters from the ranges “a-z” and “A-Z”.
`U` (Uppercase); returns 1 if *string* only contains characters from the range “A-Z”.
Note: The `datatype` method tests the meaning of the characters in a string, independent of the encoding of those characters. Extra letters and Extra digits cause `datatype` to return 0 except for the number tests (“`N`” and “`W`”), which treat extra digits whose value is in the range 0-9 as though they were the corresponding arabic numeral.
JLF : if I understand correctly, there is no support for Unicode equivalent of upper/lower case. Here, the definition is limited to "a-z" "A-Z" character set (any charset which contains these characters is supported, whatever the encoding).

nlrdef p127

d2c()
Decimal to coded character. Converts the *string* (a NetRexx *number*) to a single character, where the number is used as the encoding of the character.
string must be a non-negative whole number. An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation's encoding for characters).
Examples:
```
'77'.d2c == 'M' -- ASCII or Unicode
'+77'.d2c == 'M' -- ASCII or Unicode
'247'.d2c == '7' -- EBCDIC
'0'.d2c == '\0'
```
JLF : in ooRexx, the result is not limited to one character.
say d2c(65 * 65536 + 66 * 256 + 67) --> "ABC".
I don't see a problem here, it's just a raw encoding, byte per byte.

nlrdef p135

sequence(final)
returns a string of all characters, in ascending order of encoding, between and including the character in *string* and the character in *final*. *string* and *final* must be single characters; if *string* is greater than *final*, an error is reported.
Examples:
```
'a'.sequence('f') == 'abcdef'
'\0'.sequence('\x03') == '\x00\x01\x02\x03'
'\ufffe'.sequence('\uffff') == '\ufffe\uffff'
```

nlrdef p141

x2c()
Hexadecimal to coded character. Converts the *string* (a string of hexadecimal characters) to a single character (packs). Hexadecimal characters may be any decimal digit character (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase.
string must contain at least one hexadecimal character; insignificant leading zeros are removed, and the string is then padded with leading zeros if necessary to make a sufficient number of hexadecimal digits to describe a character encoding for the implementation.
An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation's encoding for characters).
Examples:
```
'004D'.x2c == 'M' -- ASCII or Unicode
'4d'.x2c == 'M' -- ASCII or Unicode
'A2'.x2c == 's' -- EBCDIC
'0'.x2c == '\0'
```
The `d2c` method (see page 127) can be used to convert a NetRexx number to the encoding of a character.
JLF : in ooRexx, the argument can be a string of any length. In NetRexx, the string is limited to one character. I don't see where is the problem. For me, this function iterates over the bytes of the hexadecimal string and creates the bytes of the encoded characters, one by one.

nlrsupp.pdf p10

Euro currency character
The euro character (`'\u20ac'`) is now treated in the same way as the dollar character (that is, it may be used in the names of variables and other identifiers).

It is recommended that it only be used in mechanically generated programs or where otherwise essential.
Note that only UTF8-encoded source files can currently use the euro character, and a 1.1.7 (or later) version of a Java compiler is needed to generate the class files.

nlrsupp p21

utf8
As of NetRexx 1.1 (July 1997), this option must be used as a compiler option, and applies to all programs being compiled. If present on an **options** instruction, it is checked and must match the compiler option (this allows processing with or without the **utf8** option to be enforced).

Sandbox investigations

[JLF July 23, 2010]

I think that Parrot and Ruby multi-encoding is more appropriate for ooRexx than a single Unicode wide-char encoding (less disruptive for legacy applications). The first encoding to support is UTF-8, but everything should be put in place to support other encodings.

There is an exception for oodialog, which must use the same wide-char encoding as Windows (already implemented). Note : I will have to rework the conversions to wide-char because each string will hold its own encoding. So the current implementation which depends on a global current encoding will no longer work... In the entry points (RexxMethods), the string arguments must be declared RexxStringObject instead of CSTRING. And these strings must be passed as RexxStringObject to the internal methods, to have access to the encoding informations when conversion needed (RXCA2T).

Current work : reuse the Parrot's charset/encoding implementation.

The recognised values for charset are:

- 'iso-8859-1'
- 'ascii'
- 'binary'
- 'unicode'

The encoding is implicitly guessed; Unicode implies the utf-8 encoding, and the other three assume fixed-8 encoding.

If charset is unspecified, the default charset 'ascii' is used.

Following text is borrowed from Yoko Harada's blog (Ruby M17N), adapted to an hypothetical support in ooRexx.

Script encoding

Script encoding determines an encoding of string literals in a source code. Each source file has its unique script encoding (magic comment), which is available with `.context~package~encoding`. When no magic comment is there, US-ASCII encoding is assumed.

When a script is passed through standard input, or by command-line with –e option, system locale will be applied to the script encoding only if the magic comment is missing.

The priority order of the script encoding for oorexx files :
magic comment > command-line option > US-ASCII

The priority order of the script encoding for a given script via command-line or standard input :
magic comment > command-line option > system locale

Magic Comment

The magic comment is used to specify the script encoding of a given Ruby script file. It's similar to the encoding attribute of an XML declaration in each XML file. The magic comment should be on the first line unless the script file has a shebang line. When we want to write shebang line, the magic comment comes on the second line. The format of the magic comment must match the

regular expression, /coding[:=]\s*[\w.-]+/ , which is, generally, the style of Emacs or Vim modeline.

```
#!/usr/bin/rexx
# -*- coding: utf-8 -*-
say "Emacs Style"
```

```
# vim:fileencoding=utf-8
say "Vim Style 1"
```

```
# vim:set fileencoding=utf-8 :
say "Vim Style 2"
```

```
#coding:utf-8
say "Simple"
```

The “invalid multibyte char" error will be raised when non-ASCII string literals are used in a script with no magic comment. Raising the error is good to keep platform independent of a source code. Only a script’s author knows what encoding he or she used to write the script.

Implementation note : SourceFile.cpp, `RexxSource::initBuffered :`

```
    // neutralize shell '#!...'
    if (start[0] == '#' && start[1] == '!')
    {
        memcpy(start, "--", 2);
    }
```

--> Will need to add support for encoding detection in line 1 and 2.

defaultExternal and defaultInternal encodings

`.Encoding~defaultExternal` returns the default external encoding.

external = Encoding~find("external") -- Encoding~defaultExternal is equivalent.

`.Encoding~defaultInternal` returns the default internal encoding (default is .nil).

internal = Encoding~find("internal") -- Encoding~defaultInternal is equivalent.

These encodings are used only when the encoding is not specified explicitly over standard I/O, command-line arguments, or script.

When `.Encoding~defaultInternal` is defined, the encoding of every input string is supposed to be identical to this encoding. In the same way, the encoding of strings returned from libraries are expected to be this encoding.

default_external

command-line options > locale

default_internal

command-line options > nil

Command line options: -E and -U

We can set the values of `.Encoding~defaultExternal` and `.Encoding~defaultInternal` by using –E command-line option, whose format is -Eex[:in] .

–U command-line option sets UTF-8 to both values. When we use the –U command-line option, the encoding of a script from standard input or command-line –e option is assumed to be UTF-8.

Locale Encoding

Locale encoding is determined that way :

Tries to get $LANG environment variable on both Unix and Windows to decide lthe value of localeCharmap. If $LANG variable exists, the value will be the same encoding value as $LANG.

Otherwise, tries to pick it up by GetConsoleCP on Windows (or GetACP ?).

Once the value of localeCharmap has been fixed, we can get it from `.Encoding~localeCharmap`. The locale encoding is determined from it :

Value of `.Encoding~find(Encoding~localeCharmap)`, will be US_ASCII when localeCharmap is .nil or ASCII-8BIT when localeCharmap is an unknown name. We can get the determined locale encoding by `.Encoding~find("locale")`.

The locale encoding is mainly used to set the default value of defaultExternal. Since defaultExternal is the default value of IO object's external encoding, it is applied to stdin, stdout and stderr.

ASCII Compatible Encodings

ASCII compatible encodings means that every character in the US-ASCII area is mapped to the range \x00-\x7F. We can compare or concatenate a pair of strings even though encodings of the two strings are not equivalent, under a condition : The condition is that strings to be compared and concatenated should consist of ASCII characters, and `"String"~isAsciiOnly` should return true.

```
# coding: UTF-8

a = "いろは"~encode("Shift_JIS") -- converting into Shift_JIS
say a~isAsciiOnly -- .false
b = "ABC"~encode("EUC-JP") -- converting into EUC-JP
say b~isAsciiOnly -- true

c = a || b -- "いろは ABC" -- a and b may be encoded in different encodings
say c~encoding -- Encoding:Shift_JIS
```

ASCII Incompatible Encodings

The definition of ASCII incompatible encodings is that characters in US-ASCII area are mapped to another code points of \x00-\x7F. UTF-16BE, UTF-16LE, UTF-32 BE, and UTF-32LE are categorized to this encoding.

We can't use US-ASCII incompatible encodings in a Rexx script.

We can’t concatenate with ASCII strings if the underlined strings are encoded in the ASCII incompatible ones.

Dummy Encodings

Strings of dummy encodings are byte sequences, not strings. Even though a string has ASCII characters only, comparison, concatenation, or other string operations are not supported for dummy encodings.

ISO-2022-JP, UTF-7, or other stateful encodings are in this category. They must be converted into stateless-ISO-2022-JP, or UTF-8 before using them as strings.

```
say .Encoding~ISO_2022_JP~isDummy -- .true

a = "いろは"~encode("ISO-2022-JP") -- converting into ISO-2022-JP
b = "ABC"~encode("EUC-JP") -- converting into EUC-JP
say b~isAsciiOnly -- .true
c = a || b
-- Raise error Encoding::CompatibilityError: incompatible character encodings: ISO-2022-JP and EUC-JP
```