Thoughts on 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

Updated [JLF May 27, 2012] HTML parsing : character encoding

Updated [JLF Sep 30, 2012] Python Flexible String Representation

Updated [JLF Sep 30, 2012] Article The importance of language-level abstract Unicode strings

Updated [JLF Sep 30, 2012] Twitter : counting characters

Updated [JLF Jan 06, 2013] UTR #36: Unicode Security Considerations

Updated [JLF Jan 06, 2013] Unicode Fonts for Ancient Scripts

Updated [JLF Feb 15, 2013] Character Sets

Updated [JLF Feb 16, 2013] Parrot string encodings : debug mimebase64

Updated [JLF Mar 15, 2013] Delphi and Unicode

Updated [JLF Mar 15, 2013] Go and Unicode

Updated [JLF Mar 15, 2013] .NET and Unicode

Updated [JLF Sep 02, 2013] APL

Updated [JLF Sep 08, 2013] C++

Updated [JLF Dec 22, 2013] MoarVM

Updated [JLF Oct 12, 2014] Rust & WHATWG : UTF-8 strings

Updated [JLF Nov 1, 2014] Python : experience report. Added a "conclusion" to the 1st chapter.

Updated [JLF May 14, 2015] MoarVM NFG is implemented. Red (Rebol) : Unicode issues with proposed resolutions.

Updated [JLF May 29, 2015] Swift

Updated [JLF Jul 5, 2015] Add notes to Gtk/Glib : File name encodings & checklist for application writers.

Updated [JLF Mar 26, 2016] Add a note taken from the wiki of the Red language : Normalizing a combined character to NFC then converting back to NFD does not always result in the same character sequence. One more argument to not automatically convert to Unicode when crossing the I/O barrier.

Updated [JLF Jul 30, 2016] One more article from Armin Ronacher: UCS vs UTF-8 as Internal String Encoding. Search for "lucumr.pocoo.org" to find all the links to his articles in this document. They are interesting to read.

Update [JLF Dec 10, 2016] ICU UTF-8 as main encoding : U_CHARSET_IS_UTF8

Update [JLF Dec 31, 2016] Swift: Strings, characters, and performance —a deep dive + implementation review

Update [JLF Jan 01, 2017] Python : PEP 528, Change Windows console encoding to UTF-8. PEP 529, Change Windows filesystem encoding to UTF-8, PEP 538 -- Coercing the legacy C locale to C.UTF-8.

Update [JLF Jan 01, 2017] Objecticon : maintains an index of offsets into the utf-8 string to make random access faster.

Update [JLF Apr 17,2017] WTF-8 (Wobbly Transformation Format − 8-bit).

Update [JLF May 4, 2018] Hacker News: The hell that is filename encoding (2016)

Update [JLF May 5, 2018] Python: Victor Stinner blog 3

Update [JLF Aug 4, 2019] The tragedy of UCS-2

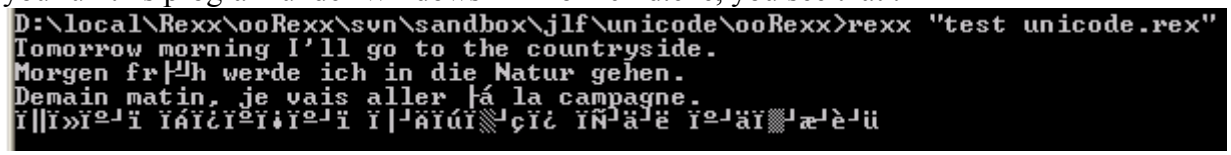
UTF-8 in ooRexx ?

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 :

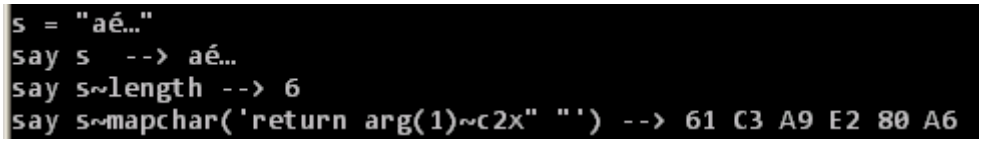


To display the character correctly in CMD shell, you need to choose the correct code page, e.g., cp65001 for UTF8 : chcp 65001
You also have to choose a font that can display the characters (e.g., Consolas or Lucida Console, NOT Raster font, for Unicode).

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.



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.
  [JLF Nov 2, 2014]4 years later : My current position is to avoid such automatic conversion... Strings entering in the ooRexx interpreter should remain unchanged. UTF-8 or UTF-16 or whatever Unicode internal encoding is not the problem. The problem is that Unicode does not support all the possible encodings, and Unicode can have different representations for a same character. So, converting back the internal string to its original encoding will not always give back the same string. The conversion must remain under control of the ooRexx programmer.

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.

[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.

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 11) : replace char by wchar everywhere in the ooRexx sources.

Probably not suitable for the whole interpreter, but should 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 is called directly, making the dialogs Unicode-enabled. The conversion must be done by ooDialog internally. That has been implemented in sandbox-jlf.

[JLF May 3, 2010] ooDialog must remain independant from ICU (unless we include ICU in the delivery). 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 pass UTF-8 string to ooDialog which converts 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.

Currently, Windows and Java use UTF-16, so if you want a direct exchange with them, without conversion, you must use UTF-16.

Cons

[JLF nov 2, 2014] From the various experience reports that I have collected, this approach is to avoid (except for ooDialog). See the conclusion at the end of this chapter.

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).

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.

Conclusion

[JLF nov 1,2014]

The systematic conversion to Unicode is to avoid. I see articles describing the problems of conversion (ex: http://lucumr.pocoo.org/2014/5/12/everything-about-unicode/). Could be ok for Windows, where Unicode is the standard behind the scene, but not ok for Linux. The thing to avoid is to ALWAYS convert to Unicode internally. From the lessons learned by others, I see that the only safe approach is to keep the strings received from the outside unchanged.

Three cases :

- The encoding of the string is unknown. This is what we have currently with ooRexx. We can use all the string methods currently available. They may return wrong (or inadapted) results but that's ok, this is the legacy behaviour. The Unicode services can't be used on this string. If the ooRexx programmer wants to use the Unicode services, then he will have to convert the string to Unicode, by providing an encoding. If the conversion fails, then end of the story. If the conversion succeeds then both versions are available (original string and Unicode string). It's up to the ooRexx programmer to decide if he wants to work now only with the Unicode version, or if the original string is still needed. An example where the original string is still needed : you get a list of filenames, you want to convert them to Unicode for presentation purpose, but later you will need to pass again these filenames to the operating system. Converting back from unicode will SOMETIMES give you a string which is different from the original string ! The only safe way is to pass the original string.
- The encoding of the string is known, and is not Unicode : Same as previous case, just store the encoding on the string. To use the Unicode services, the ooRexx programmer will have to convert the string to Unicode first.

- The encoding of the string is known, and is Unicode. Store the encoding on the string. You have access to all the Unicode services. [JLF Mar 26, 2016] ~~Here, the interpreter can decide to convert the string to an internal format well suited for the Unicode services, which is different from the original encoding. That's ok because it will be possible to convert back to the original encoding, without lossing characters. (well... to check : is there a risk of no longer finding a file after normalization ?)~~ Normalizing a combined character to NFC(Normal Form Composed) form, then converting the result back to NFD (Normal Form Decomposed) form does not always result in the same character sequence. The Unicode standard maintains a list of exceptions: http://www.unicode.org/Public/UCD/latest/ucd/CompositionExclusions.txt. Characters on this list are composable, but not decomposable back to their combined form, for various reasons. Also see the documentation on the Composition Exclusion Table: http://www.unicode.org/reports/tr15/#Primary_Exclusion_List_Table

So far, the key point is that there is no automatic conversion to Unicode. And that must stay like that, even when concatenating strings with different encodings. That can be achieved using Ropes (see Boehm's paper). A Rope is an ordered tree, with each internal node representing the concatenation of its childrens, and the leaves consisting of flat strings. A leaf string has a single encoding.

From the ooRexx programmer perspective :

- A new class Text (?) must be provided, which exposes an interface which is applicable to any string, whatever its encoding (UTF-8, UTF-16, ...). This interface should be similar to the String interface, but here the indexes are graphemes indexes, not bytes indexes.
- The String class remains as-is, not encoding-aware, byte-oriented.
- aText~string returns its buffer as a String object. There is no hidden conversion here.
- A Text object can be created from a String object, passing an optional encoding.
  Ex : t1 = .Text~new("hello") – default encoding
  Ex : t2 = .Text~new("hello", "utf-8") – utf-8 encoding
  Ex : t3 = .Text~new("hello"||"e280a6"x, "utf-8") – "hello…"
  "…" is 3 bytes because the utf-8 encoding uses 3 bytes, not because you see 3 dots. This is really ONE unicode character (HORIZONTAL ELLIPSIS, hex codepoint 2026).
  Ex : t4 = .Text~new("hello") || .Text~new("e280a6"x, "utf-8") – same as t3

JLF 11/09/2016 : Several classic Rexx scripts in RosettaCode contain UTF-8 characters. Ex: in Aliquot-sequence-classifications, there is

```
say center('numbers from ' low " to " high, 79, "═")
```

where "═" is the Unicode character BOX DRAWINGS DOUBLE HORIZONTAL, encoded E2 95 90 in UTF-8. ooRexx and Regina complains:

Error 40.23: CENTER argument 3 must be a single character; found "═"

I would like an intelligent support of UTF-8 here, without modifying the Rexx script. The length in grapheme is 1, the interpreter should not complain, even if the length in bytes is <> 1.

Side note : A Rope is an immutable date structure. Ropes have been in use in the Cedar environment almost since its inception. Clojure and Haskell implement immutable (persistent) data structures.

https://github.com/ivmai/bdwgc/tree/master/cord

http://en.wikipedia.org/wiki/Rope_(data_structure))

http://staff.city.ac.uk/~ross/papers/FingerTree.html

https://github.com/clojure/data.finger-tree

https://github.com/e-user/engine/blob/master/src/engine/data/rope.clj

http://www.ibm.com/developerworks/library/j-ropes/

http://en.wikipedia.org/wiki/Radix_tree

See https://mail.mozilla.org/pipermail/rust-dev/2014-May/009725.html

for a discussion about ropes in Rust.

The begining of the discussion is focused on the "wrong" design of ruby 1.9.

In conclusion, the idea of ropes in the stdlib is rejected. They prefer to convert early (I/O barrier) rather than converting later when concatenating strings of different encodings.

JLF : that's correct, but this is because they insist on converting to a unique internal encoding.

See https://news.ycombinator.com/item?id=7605833

for a little discussion about ropes.

One of the opinions is : "what's important is that handling of encoding happens on the I/O barrier, and not anywhere else."

JLF : that's correct, but this is because they insist on converting to a unique internal encoding.

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.

[icu-support] UTF-8 as main encoding

[JLF Dec 10, 2016]

https://sourceforge.net/p/icu/mailman/message/32031609/

> Is there any limitation that we should know by using U_CHARSET_IS_UTF8?

You should use it if your platform is known to use UTF-8 as its system charset, such as MacOS X or a modern Linux distribution.

You cannot use it if your platform uses a different system charset, or a variety of them, such as Windows.

See http://userguide.icu-project.org/strings/utf-8

Note: This build-config option only routes ICU char* APIs from "use the default converter" to "assume char* is UTF-8", which improves performance and reduces intra-library dependencies. It changes nothing about ICU's processing of UTF-16 strings in most of its code (with the exceptions of UTF-8 or text-interface APIs).

> My hope is that we can cover most of the core operations without relying on UTF-16 processing. As you suggested, if something is missing we can always try to implement our own functions, which could be contributed back to ICU if desired. After having read many discussions about the need of a general UTF-8 processing library, it could be a good idea to think about adding more UTF-8 specific functions to ICU.

FYI, for performance, there is a trade-off between staying in UTF-8 because your input/output data is in UTF-8, and converting to UTF-16 because it can be much faster to process. When and where you care about performance, you should measure rather than make assumptions. When you don't, do what's convenient, easy, and easy to maintain.

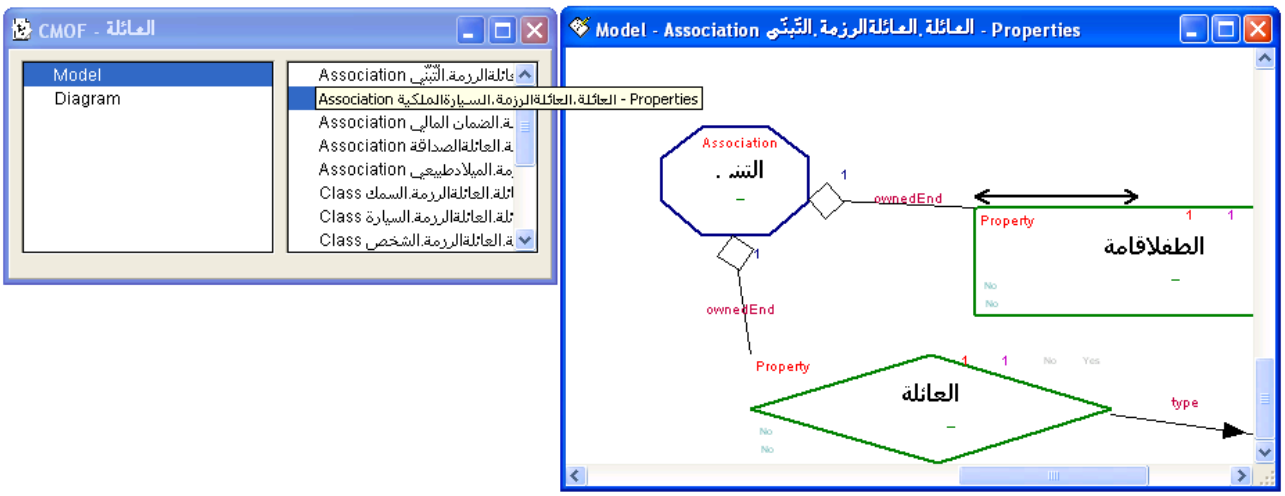
For example, I recently rewrote the ICU collation code, and I made equivalent specially-optimized fastpaths for both UTF-8 and UTF-16 in performance-sensitive areas, with shared code (operating on code points) for most else. I was quite surprised to find that sorting and binary search was at least 3x as fast with UTF-16 as with UTF-8 (using the same input).

See "UTF-8 vs. UTF-16" on http://site.icu-project.org/design/collation/v2/perf

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/resources/odbc/unicode/odbc-driver.html

APL

Unicode and related subjects in APL2

http://archive.vector.org.uk/art10500670

Extract :

Microsoft Windows offers a Unicode font called Arial Unicode MS. This font we found is not suitable to write APL functions with, because of the unfamiliar shapes and unbalanced sizes of its APL characters and pitches. New system font Courier APL2 Unicode on the other hand behaved as if it contained Japanese characters as well under Windows XP and Vista, and we found it a perfect font for our use.

Unicode support for APL (Dyalog)

http://archive.vector.org.uk/art10500090

Historically, APL systems have used a single byte (8 bits) to represent each character in an array. The list of all the 256 possible characters [1] is known to APL users as the Atomic Vector, a system variable named ⎕AV. Atomic vectors differ from one vendor to the next, sometimes even between products from the same vendor, and some vendors (including Dyalog) allow the user to redefine sections of this system constant, in order to support different national languages. Obviously, a single byte per character is no longer sufficient to represent character arrays in a Unicode APL system.

IBM added multi-byte character support to APL2 before the first version of the Unicode standard saw the light of day. From Version 1 Release 3 (dated 1987), APL2 uses 1 and 4 byte representations for character data. The 1-byte representation is used for arrays containing characters that are all elements of the APL2 atomic vector. The 4-byte representation is used for other character arrays. In the 4 byte representation, 2 bytes contain the data's codepage and 2 bytes contain the data's code point. Support for Unicode was added in Version 2 Release 2 (dated 1994), where APL2 supports codepage values of zero and 1200 which both indicate the data uses the UCS-2 representation. These APL2 representations have the advantages that existing applications can run without change and applications which only use characters in the APL2 atomic vector have minimal storage requirements.

J introduced a double-byte character type containing UCS-2 encoded characters in Version 4.06, which was released in 2001. From Version 6.01 (September 2006), J switched from using ANSI to UTF-8 as the default encoding of (single-byte) character constants entering the J system from the keyboard and through the execution of J script files. This change was made in order to make it easy to work with text files, and in particular to support J scripts containing Unicode characters, as UTF-8 had emerged as the de facto standard for such files. If J applications wish to view Unicode strings as arrays where each element of the array is exactly one Unicode character, conversion to the double-byte (UCS-2) representation is required.

After much debate, we decided that the new Unicode product should only have a single type of character as seen from the user's point of view. Therefore, all character arrays contain one Unicode

code point number for each character in the array. The system picks the smallest possible internal representation (1, 2 or 4-bytes), depending on the range of elements in the array – in the same way that Dyalog APL has always stored integers. This means that most existing customer data will continue to be represented using one byte per character. Arrays which contain APL symbols require two bytes per element, and a very small number of characters require four bytes (but no pre-existing data created by an earlier version of the system will require more than two).

⎕UCS is a new system function for converting to and from integer code point numbers to the corresponding characters.

```
      ⎕UCS'こんにちは世界'
12371 12435 12395 12385 12399 19990 30028
```

The monadic function is self-inverse, and has the same definition as in APL2 implementation: a right argument of code points gives you a character array of the same shape:

```
      ⎕UCS 123 9077 91 9035 9077 93 125
{ω[⎕ω]}
```

In Dyalog APL, ⎕UCS also accepts a left argument which must be the name of a UTF encoding. In this case, if the right argument is a character array, the numeric result contains the encoded data stream, and vice versa:

```
      'UTF-8' ⎕UCS '界'
231 149 140
      'UTF-8' ⎕UCS 65 195 132
AÄ
```

The dyadic extension makes it straightforward to work with variable-length encodings. Writing any character string to a UTF-8 encoded text file requires a slightly more complex expression, along the lines of:

```
(⎕UCS 'UTF-8' ⎕UCS text) ⎕NAPPEND tn
```

The leftmost ⎕UCS converts the integers returned by dyadic ⎕UCS back into characters before writing them to file. This is necessary because numbers in the range 128-255 would cause data to be written using two bytes per element. In effect, the last ⎕UCS is used to turn the numbers into 8-bit unsigned integers. We debated adding an ‘unsigned’ type number to native file functions, or even a type number to select UTF-8 encoding as part of the native file interface, but decided to be conservative in the first Unicode release, as the above expression is quite straightforward and easily embedded in utilities for manipulating files.

If your application inhabits the Microsoft.Net framework, you can leave the encoding to the framework, and simply write:

```
System.IO.File.WriteAllText filename text
```

The default encoding used is UTF-8, but this can also be selected using an optional third argument, for example:

```
System.Text.Encoding.UTF16
```

C++

Unicode support in the standard library.

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3572.html

reddit

http://www.reddit.com/r/programming/comments/1lq82r/c_proposal_to_add_unicode_support_in_the_standard/

Committee feedback on N3572

https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/9wl0dNF818Q%5B1-25-false%5D

Character Sets

[JLF Feb 15, 2013]

IANA

http://www.iana.org/assignments/character-sets/character-sets.xml

Official names for character sets.

http://www.firstobject.com/character-set-name-alias-code-page.htm
Character set (charset) name, alias and code page

Delphi

[JLF Mar 15,2013]

Delphi 2009 fully supports Unicode. They've changed the implementation of `string` to default to 16-bit Unicode encoding, and most libraries including the third party ones support Unicode. See Marco Cantù's Delphi and Unicode. [JLF Mar 15, 2013] This is worth reading.

Prior to Delphi 2009, the support for Unicode was limited, but there was `WideChar` and `WideString` to store the 16-bit encoded string. See Unicode in Delphi for more info.

Note, you can still develop bilingual CJKV application without using Unicode. For example, Shift JIS encoded string for Japanese can be stored using plain `AnsiString`.

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).

Forth

http://www.forth200x.org/xchar.html

Fonts

http://users.teilar.gr/~g1951d/

Unicode Fonts for Ancient Scripts : Aegean and Mediterranean Scripts, Egyptian Hieroglyphs,

Symbola, fonts based on early editions of Greek texts, et al.

http://unifoundry.com/unifont.html

Unifoundry.com

GNU Unifont Glyphs

GTK

[JLF May 4, 2010]

Uses UTF-8 internally.

Unicode Manipulation

https://developer.gnome.org/glib/stable/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.

File Name Encodings

[JLF Jul 5, 2015]

https://developer.gnome.org/glib/stable/glib-Character-Set-Conversion.html

Historically, **Unix has not had a defined encoding for file names**: a file name is valid as long as it does not have path separators in it ("/"). However, displaying file names may require conversion: from the character set in which they were created, to the character set in which the application operates.

Glib uses UTF-8 for its strings, and GUI toolkits like GTK+ that use Glib do the same thing. If you get a file name from the file system, for example, from readdir(3) or from g_dir_read_name(), and you wish to display the file name to the user, you will need to convert it into UTF-8. The opposite case is when the user types the name of a file he wishes to save: the toolkit will give you that string in UTF-8 encoding, and you will need to convert it to the character set used for file names before you can create the file with open(2) or fopen(3).

Glib uses UTF-8 for its strings, and GUI toolkits like GTK+ that use Glib do the same thing. If you get a file name from the file system, for example, from readdir(3) or from g_dir_read_name(), and you wish to display the file name to the user, you will need to convert it into UTF-8. The opposite case is when the user types the name of a file he wishes to save: the toolkit will give you that string in UTF-8 encoding, and you will need to convert it to the character set used for file names before you can create the file with open(2) or fopen(3).

By default, Glib assumes that file names on disk are in UTF-8 encoding. This is a valid assumption for file systems which were created relatively recently: most applications use UTF-8 encoding for their strings, and that is also what they use for the file names they create. However, older file systems may still contain file names created in "older" encodings, such as ISO-8859-1. In this case, for compatibility reasons, you may want to instruct Glib to use that particular encoding for file names rather than UTF-8. You can do this by specifying the encoding for file names in the G_FILENAME_ENCODING environment variable. For example, if your installation uses ISO-8859-1 for file names, you can put this in your ~/.profile:

export G_FILENAME_ENCODING=ISO-8859-1

Glib provides the functions g_filename_to_utf8() and g_filename_from_utf8() to perform the necessary conversions. These functions convert file names from the encoding specified in G_FILENAME_ENCODING to UTF-8 and vice-versa.

Checklist for Application Writers

[JLF Jul 5, 2015]

https://developer.gnome.org/glib/stable/glib-Character-Set-Conversion.html

This section is a practical summary of the detailed description above. You can use this as a checklist of things to do to make sure your applications process file name encodings correctly.

1. If you get a file name from the file system from a function such as readdir(3) or gtk_file_chooser_get_filename(), you do not need to do any conversion to pass that file name to functions like open(2), rename(2), or fopen(3) — those are "raw" file names which the file system understands.
2. If you need to display a file name, convert it to UTF-8 first by using g_filename_to_utf8(). If conversion fails, display a string like "Unknown file name". Do not convert this string back into the encoding used for file names if you wish to pass it to the file system; use the original file name instead. For example, the document window of a word processor could display "Unknown file name" in its title bar but still let the user save the file, as it would keep the raw file name internally. This can happen if the user has not set the G_FILENAME_ENCODING environment variable even though he has files whose names are not encoded in UTF-8.
3. If your user interface lets the user type a file name for saving or renaming, convert it to the encoding used for file names in the file system by using g_filename_from_utf8(). Pass the

converted file name to functions like fopen(3). If conversion fails, ask the user to enter a different file name. This can happen if the user types Japanese characters when G_FILENAME_ENCODING is set to ISO-8859-1, for example.

Go

[JLF Mar 15, 2013]

http://golang.org/ref/spec

Source code representation

Source code is Unicode text encoded in UTF-8. The text is not canonicalized, so a single accented code point is distinct from the same character constructed from combining an accent and a letter; those are treated as two code points. For simplicity, this document will use the unqualified term *character* to refer to a Unicode code point in the source text.

Each code point is distinct; for instance, upper and lower case letters are different characters.

Implementation restriction: For compatibility with other tools, a compiler may disallow the NUL character (U+0000) in the source text.

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes. Within the quotes, any character may appear except single quote and newline. A single quoted character represents the Unicode value of the character itself, while multi-character sequences beginning with a backslash encode values in various formats.

The simplest form represents the single character within the quotes; since Go source text is Unicode characters encoded in UTF-8, multiple UTF-8-encoded bytes may represent a single integer value. For instance, the literal `'a'` holds a single byte representing a literal `a`, Unicode U+0061, value `0x61`, while `'ä'` holds two bytes (`0xc3 0xa4`) representing a literal a-dieresis, U+00E4, value `0xe4`.

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

Raw string literals are character sequences between back quotes ``` ` ```. Within the quotes, any character is legal except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage returns inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes `""`. The text between the quotes, which may not contain newlines, forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that `\'` is illegal and `\"` is legal), with the same restrictions. The three-digit octal (`\`*nnn*) and two-digit hexadecimal (`\x`*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal `\377` and `\xFF` represent a single byte of value `0xFF`=255, while ÿ, `\u00FF`, `\U000000FF` and `\xc3\xbf` represent the two bytes `0xc3 0xbf` of the UTF-8 encoding of character U+00FF.

For statements

For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of

successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.

Hacker News

The hell that is filename encoding (2016)

https://news.ycombinator.com/item?id=16991263
Like on Unix, Windows does not require a path to be valid Unicode.
That is, on Windows, paths are fundamentally sequences of 16-bit words, just like on Unix paths are fundamentally sequences of 8-bit bytes. On neither system are paths fundamentally text.

The story then goes on further: every NTFS volume contains a special file named `$UpCase` that has a uppercase mapping for all possible 16-bit words, resulting in an 128 KiB table. This approach has an upside for backward and forward compatibility... unless you eventually need a case mapping for non-BMP characters or complex mapping that expands to multiple characters.

NTFS is (usually) case preserving but not case-sensitive. So the OS needs to be able to tell whether EXAMPLE.TXT and example.txt are the "same" name, which means it needs case conversion.

Not everybody agrees about how this conversion should work. The most famous example is Turkish, but there are others. So there's an actual choice to make here.

If Windows baked this into the core OS, they might get pushback in countries where their (presumably American) defaults were culturally unacceptable.

If they made it configurable at the OS level, everything would seem fine until, say, a German tries to access a USB drive with files from a Turk on them and some files don't work correctly, or the disk just can't be mounted at all.

So, they have to bake it into each NTFS filesystem.

The tragedy of UCS-2

https://news.ycombinator.com/item?id=20600195

cesarb 2 hours ago [-]

Another aspect of this tragedy is that the creation of UTF-16 has forever limited Unicode to only 17 planes. The original UTF-8 encoding (https://tools.ietf.org/html/rfc2044) used up to six bytes, and could represent a full 31-bit range, which corresponds to 32768 16-bit planes; after UTF-16 became common, UTF-8 was limited to four bytes (https://tools.ietf.org/html/rfc3629).

kps 2 hours ago [-]

The original pattern extends up to at least 36 bits (which makes the PDP-10 refugees happy).

1 7 0xxxxxxx

2 11 110xxxxx 10xxxxxx

3 16 1110xxxx 10xxxxxx 10xxxxxx

4 21 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

5 26 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

6 31 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

7 36 11111110 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Then, you can either decide to limit the leader to one byte, giving you 42 bits…

8 42 11111111 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Or allow multi-byte leaders and continue forever.

9 53 11111111 110xxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

10 58 11111111 1110xxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

⋮

The Intergalactic Confederation will not look kindly on UTF-16.

cesarb 2 hours ago [-]

> Or allow multi-byte leaders and continue forever.

That loses one very useful property of UTF-8: you always know, by looking at a single byte, whether it's the first byte of a code point or not. It also loses the property that no valid UTF-8 encoding is a subset of another valid UTF-8 encoding. It's better to stop at 36 bits (which keeps another useful property: you'll never find a 0xFF byte within an UTF-8 string, and you'll only find a 0x00 byte when it's encoding the U+0000 code point).

cat199 2 hours ago [-]

> The original pattern extends up to at least 36 bits (which makes the PDP-10 refugees happy).

Naah.. there's UTF-9 and UTF-18 for that..

https://tools.ietf.org/html/rfc4042

userbinator 1 hour ago [-]

Why would you ever need 31 bits? That's over TWO BILLION distinct codepoints.

Only ~137K have been assigned so far, or in other words a little over 17 bits. Given that each additional bit doubles the range, even in the case that characters continue to be assigned at a constant rate, reaching that high is so far into the future that it's not worth thinking about. Unicode is around 30 years old; at the same rate, another 30 years would be needed to get to 18 bits, then another 60, 120, ...

On the other hand, it is extremely wasteful to burden all UTF-8 processing code with handling values that will literally never be encountered outside of error cases.

amluto 2 hours ago [-]

Is this really forever? ISTM the old version could be reintroduced as UTF-8+ without insurmountable problems and with a good degree of compatibility.

Haskell

Bindings to the ICU library

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

HTML parsing

[JLF May 27, 2012]

https://html.spec.whatwg.org/multipage/syntax.html#parsing

Interesting section about character encoding.

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

Java™ Internationalization Support

http://docs.oracle.com/javase/7/docs/technotes/guides/intl/index.html

Every implementation of the Java platform is required to support the following standard charsets. Consult the release documentation for your implementation to see if any other charsets are supported. The behavior of such optional charsets may differ between implementations.

Charset	Description
`US-ASCII`	Seven-bit ASCII, a.k.a. `ISO646-US`, a.k.a. the Basic Latin block of the Unicode character set
`ISO-8859-1`	ISO Latin Alphabet No. 1, a.k.a. `ISO-LATIN-1`
`UTF-8`	Eight-bit UCS Transformation Format
`UTF-16BE`	Sixteen-bit UCS Transformation Format, big-endian byte order
`UTF-16LE`	Sixteen-bit UCS Transformation Format, little-endian byte order
`UTF-16`	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Every instance of the Java virtual machine has a default charset, which may or may not be one of the standard charsets. The default charset is determined during virtual-machine startup and typically depends upon the locale and charset being used by the underlying operating system.

http://illegalargumentexception.blogspot.co.uk/2009/04/java-unicode-on-windows-command-line.html

Java: Unicode on the Windows command line

http://illegalargumentexception.blogspot.co.uk/2009/05/java-rough-guide-to-character-

encoding.html

Java: a rough guide to character encoding

http://illegalargumentexception.blogspot.co.uk/2009/12/java-safe-character-handling-and-url.html

Java: safe character handling and URL building

http://illegalargumentexception.blogspot.co.uk/2013/06/java-detecting-json-character-encoding.html

Java: detecting JSON character encoding

[JLF Aug 4, 2019]

"Compact strings" (https://openjdk.java.net/jeps/254) shipped with JDK 9 in September 2017 (https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID...)

JS, like Java, now has implementations that will also store strings as Latin1 when the implementation believes it is safe to do so. This results in significant memory savings [1] in most programs. https://blog.mozilla.org/javascript/2014/07/21/slimmer-and-f...

JavaScript

[JLF Aug 4, 2019]

Jan de Mooij wrote on July 22, 2014 at 8:57 am:

As far as I know, all JS engines, including SpiderMonkey, treat JS strings as a sequence of 16-bit "integers" (code units). So strings are not fully (variable-length) UTF16 and character indexing is O(1). See also http://www.2ality.com/2013/09/javascript-unicode.html

Simon Oram wrote on July 22, 2014 at 10:31 pm :

Yes, this is the behavior required by the specification (and sadly it cannot be changed without breaking the Web). See also JavaScript has a Unicode problem and JavaScript's internal character encoding: UCS-2 or UTF-16?.

Lisp

https://lists.clozure.com/pipermail/openmcl-devel/2007-April/003206.html

Clozure : How many angels can dance on a unicode character ?

Note JLF : discussion about the best size for char... Today, Clozure use UTF-32.

https://groups.google.com/forum/?fromgroups=#!topic/comp.lang.lisp/l_8QOu52raI

Unicode and Common Lisp

http://www.franz.com/support/documentation/9.0/doc/iacl.htm

International Character Support in Allegro CL

Lua

http://lua-users.org/wiki/LuaUnicode

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

Unicode and MBCS

https://msdn.microsoft.com/en-us/library/cwe8bzh0.aspx

Support for Unicode

https://msdn.microsoft.com/en-us/library/2dax2h36.aspx

Support for Multibyte Character Sets (MBCSs)

https://msdn.microsoft.com/en-us/library/5z097dxa.aspx

Generic-Text Mappings in Tchar.h

https://msdn.microsoft.com/en-us/library/c426s321.aspx

How to: Convert Between Various String Types

https://msdn.microsoft.com/en-us/library/ms235631.aspx

MoarVM

[JLF May 14, 2015]

NFG is implemented.

http://moarvm.com/index.html

MoarVM 2015.04

Implementation of Unicode Normalization (NFC, NFD, NFKC, NFKD), and a partial implementation of Normal Form Grapheme.

Unicode Database upgraded to version 7.0.

https://6guts.wordpress.com/2015/04/30/this-week-the-big-nfg-switch-on-and-many-fixes/

Blog of Jonathan Worthington, the guy who implemented NFG for Perl6.

Rakudo on MoarVM uses NFG now

Since the 24th April, all strings you work with in Rakudo on MoarVM are at grapheme level.

[JLF Dec 22, 2013]

https://github.com/MoarVM/MoarVM

http://6guts.wordpress.com/2013/05/31/moarvm-a-virtual-machine-for-nqp-and-rakudo/

Unicode strings, designed with future NFG support in mind. The VM includes the Unicode Character Database, meaning that character name and property lookups, case changing and so forth can be supported without any external dependencies. Encoding of strings takes place only at the point of I/O or when a Buf rather than a Str is requested; the rest of the time, strings are opaque (we're working towards NFG and also ropes).

http://jnthn.net/papers/2013-yapceu-moarvm.pdf

MoarVM includes the Unicode Character Database so far as we need it for Perl 6.

No external dependencies (like ICU).

Rather well compressed; even with all of this included, the full MoarVM executable weighs in at ~ 2.5 MB.

Support the various case change operations, character property lookups (also used for regex character classes), character name resolution..

NFC (Normalization Form C) will always collapse a codepoint followed by a combining codepoint into a single codepoint if one is available :

o (U+006F) + ¨(U+0308) --> ö (U+00F6)

NGF (G = Grapheme) takes it a step further; if a single code point is not available, it makes one up (relying on being able to use negative integers to represent these). This means we can treat even strings with combining characters as fixed width and get things right.

http://rdstar.wordpress.com/2013/07/22/some-thoughts-on-unicode-in-perl-6/

https://github.com/perl6/specs/blob/master/S15-unicode.pod

.NET

[JLF Mar 15,2013]

http://msdn.microsoft.com/en-us/library/system.string.length.aspx

The Length property returns the number of Char objects in this instance, not the number of Unicode characters. The reason is that a Unicode character might be represented by more than one Char. Use the System.Globalization.StringInfo class to work with each Unicode character instead of each Char.

http://msdn.microsoft.com/en-us/library/system.globalization.stringinfo.aspx

Provides functionality to split a string into text elements and to iterate through those text elements.

The .NET Framework defines a text element as a unit of text that is displayed as a single character, that is, a grapheme. A text element can be a base character, a surrogate pair, or a combining character sequence. The Unicode Standard defines a surrogate pair as a coded character representation for a single abstract character that consists of a sequence of two code units, where the first unit of the pair is a high surrogate and the second is a low surrogate. The Unicode Standard defines a combining character sequence as a combination of a base character and one or more combining characters. A surrogate pair can represent a base character or a combining character.

ObjectIcon

[JLF Jan 01, 2017]

http://objecticon.sourceforge.net/Unicode.html

ucs (standing for Unicode character string) is a new builtin type, whose behaviour closely mirrors that of the conventional Icon string. It operates by providing a wrapper around a conventional conventional Icon string, which must be in utf-8 format. This has several advantages, and only one serious disadvantage, namely that a utf-8 string is not randomly accessible, in the sense that one cannot say where the representation for unicode character i begins. To alleviate this disadvantage, the ucs type maintains an index of offsets into the utf-8 string to make random access faster. The size of the index is only a few percent of the total allocation for the ucs object.

Oracle

http://docs.oracle.com/cd/A91202_01/901_doc/server.901/a90236/ch5.htm

Supporting Multilingual Databases with Unicode.

UnicodeOutOfTheBoxTest

[May 5, 2018. Last commit Feb 25, 2015]

https://github.com/PeterWAWood/UnicodeOutOfTheBoxTests

A short set of tests to give an indication of how well a language supports Unicode "Out of the Box". These tests only address basic string features and not text processing features such as end of word and paragraph support.

Langugaes: C#, Cocoa, Factor, Go, Java, JavaScript, LiveCode7, PHP, Perl, Python3, Racket, Rebol2, Rebol3, Red042, Ruby.

Parrot

[JLF May 27, 2010]

Essay

What the heck is: A string (oct 11 2003)

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

Strings, some practical advice (oct 14 2003)

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

The Pain of Text (jan 20 2004)

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

Parrot strings & Grapheme Normalization Form (NFG)

JLF : careful ! NFG is not part of the Unicode standard. Moreover, there is no implementation of NFG in Parrot... But the idea is interesting.

[JLF May 14, 2015] NFG is implemented in MoarVM.

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.

How to debug a string encoding problem

[JLF Feb 16, 2013] Parrot string encodings : debug mimebase64

http://perl6advent.wordpress.com/2012/12/07/day-7-mimebase64-on-encoded-strings/

PERL

[JLF May 27, 2010]

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

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

http://stackoverflow.com/questions/6162484/why-does-modern-perl-avoid-utf-8-by-default
(see answer from Perl guru Tom Christiansen)

http://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html
List of 44 recipes for working with Unicode in Perl 5 (by Perl guru Tom Christiansen)

http://rdstar.wordpress.com/2013/07/22/some-thoughts-on-unicode-in-perl-6/
Some Thoughts on Unicode in Perl 6

https://en.wikibooks.org/wiki/Perl_Programming/Unicode_UTF-8
Perl Programming/Unicode UTF-8

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

http://www.slideshare.net/andreizm/the-good-the-bad-and-the-ugly-what-happened-to-unicode-and-php-6

http://lwn.net/Articles/379909/

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

Plan 9

Plan 9 was the first operating system with complete support for the UTF-8 Unicode character set encoding.

http://plan9.bell-labs.com/sys/doc/utf.html

http://swtch.com/plan9port/

The UTF-8 library, the formatted print library, the buffered I/O library, the (Unicode-capable) regular expression library, and mk are available in packaging separate from plan9port.

See http://swtch.com/plan9port/unix/

rune - rune/UTF conversion

```
runetochar, chartorune, runelen, runenlen, fullrune, utfecpy,
utflen, utfnlen, utfrune, utfrrune, utfutf
```

Python

Updated [JLF Sep 30, 2012]

Various links

http://www.python.org/dev/peps/pep-0393/

Flexible String Representation
--> similar to Falcon approach

http://docs.python.org/release/3.2/whatsnew/3.0.html#text-vs-data-instead-of-unicode-vs-8-bit

http://docs.python.org/dev/howto/unicode.html

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

Non-decodable Bytes in System Character Interfaces

http://legacy.python.org/dev/peps/pep-0383/

With this PEP, non-decodable bytes >= 128 will be represented as lone surrogate codes U+DC80..U+DCFF. Bytes below 128 will produce exceptions.

To convert non-decodable bytes, a new error handler "surrogateescape" is introduced, which produces these surrogates. On encoding, the error handler converts the surrogate back to the corresponding byte. This error handler will be used in any API that receives or produces file names, command line arguments, or environment variables.

Unicode in Python 2 and Python 3

http://lucumr.pocoo.org/2014/1/5/unicode-in-2-and-3/

Interesting critics !

An interesting story about the difficulties to implement the 'cat' command with Python 3, because of the decision to encode internally to Unicode.

http://lucumr.pocoo.org/2014/5/12/everything-about-unicode/

The troubles with Unicode support in Python 3 (by Armin Ronacher)

http://click.pocoo.org/5/python3/

These troubles have been taken into account in PEP 538.

Victor Stinner blog 3

[JLF May 5, 2018]

https://vstinner.github.io/category/python.html

Mar 27, 2018 - Python 3.7 UTF-8 Mode

https://vstinner.github.io/python37-new-utf8-mode.html

UTF-8 is the best encoding in most cases, but it is still not the best encoding in all cases, even in 2018. The locale encoding remains the best default filesystem encoding for Python. I would say that the locale encoding is the least bad filesystem encoding.

This article tells the story of my PEP 540: Add a new UTF-8 Mode which adds an opt-in option to "use UTF-8" everywhere". Moreover, the UTF-8 Mode is enabled by the POSIX locale: Python 3.7 now uses UTF-8 for the POSIX locale. My PEP 540 is complementary to Nick Coghlan's PEP 538.

PEP

PEP 383 -- Non-decodable Bytes in System Character Interfaces

[JLF May 6, 2018]

PEP created 22-Apr-2009

File names, environment variables, and command line arguments are defined as being character data in POSIX; the C APIs however allow passing arbitrary bytes - whether these conform to a certain encoding or not. This PEP proposes a means of dealing with such irregularities by embedding the bytes in character strings in such a way that allows recreation of the original byte string.

PEP 528, Change Windows console encoding to UTF-8

[JLF Jan 01, 2017]

https://www.python.org/dev/peps/pep-0528/

Historically, Python uses the ANSI APIs for interacting with the Windows operating system, often via C Runtime functions. However, these have been long discouraged in favor of the UTF-16 APIs. Within the operating system, all text is represented as UTF-16, and the ANSI APIs perform encoding and decoding using the active code page.

This PEP proposes changing the default standard stream implementation on Windows to use the Unicode APIs. This will allow users to print and input the full range of Unicode characters at the default Windows console. This also requires a subtle change to how the tokenizer parses text from readline hooks.

PEP 529, Change Windows filesystem encoding to UTF-8

[JLF Jan 01, 2017]

https://www.python.org/dev/peps/pep-0529/

This PEP proposes changing the default filesystem encoding on Windows to utf-8, and changing all filesystem functions to use the Unicode APIs for filesystem paths. This will not affect code that uses strings to represent paths, however those that use bytes for paths will now be able to correctly round-trip all valid paths in Windows filesystems. Currently, the conversions between Unicode (in the OS) and bytes (in Python) were lossy and would fail to round-trip characters outside of the user's active code page.

Notably, this does not impact the encoding of the contents of files. These will continue to default to locale.getpreferredencoding() (for text files) or plain bytes (for binary files). This only affects the encoding used when users pass a bytes object to Python where it is then passed to the operating system as a path name.

This proposal would remove all use of the *A APIs and only ever call the *W APIs. When Windows returns paths to Python as str , they will be decoded from utf-16-le and returned as text (in whatever

the minimal representation is). When Python code requests paths as bytes , the paths will be transcoded from utf-16-le into utf-8 using surrogatepass (Windows does not validate surrogate pairs, so it is possible to have invalid surrogates in filenames). Equally, when paths are provided as bytes , they are transcoded from utf-8 into utf-16-le and passed to the *W APIs.

PEP 538 -- Coercing the legacy C locale to C.UTF-8

[JLF Jan 01, 2017]

https://www.python.org/dev/peps/pep-0538/

This PEP proposes that the CPython implementation be changed such that:

- when used as a library, Py_Initialize will warn that use of the legacy C locale may cause various Unicode compatibility issues : *Py_Initialize detected LC_CTYPE=C, which limits Unicode compatibility. Some libraries and operating system interfaces may not work correctly. Set `PYTHONALLOWCLOCALE=1 LC_CTYPE=C` to configure a similar environment when running Python directly*.
- when used as a standalone binary, CPython will automatically coerce the C locale to C.UTF-8 unless the new PYTHONALLOWCLOCALE environment variable is set : *Python detected LC_CTYPE=C, forcing LC_ALL & LANG to C.UTF-8 (set PYTHONALLOWCLOCALE to disable this locale coercion behaviour)*.

With this change, any *nix platform that does not offer the C.UTF-8 locale as part of its standard configuration will only be considered a fully supported platform for CPython 3.7+ deployments when a non-ASCII locale is set explicitly.

Rationale:

It has been clear for some time that the C locale's default encoding of ASCII is entirely the wrong choice for development of modern networked services. Newer languages like Rust and Go have eschewed that default entirely, and instead made it a deployment requirement that systems be configured to use UTF-8 as the text encoding for operating system interfaces. Similarly, Node.js assumes UTF-8 by default (a behaviour inherited from the V8 JavaScript engine) and requires custom build settings to indicate it should use the system locale settings for locale-aware operations.

Dropping official support for Unicode handling in the legacy C locale:

We've been trying to get strict bytes/text separation to work reliably in the legacy C locale for over a decade at this point. Not only haven't we been able to get it to work, neither has anyone else - the only viable alternatives identified have been to pass the bytes along verbatim without eagerly decoding them to text (Python 2, Ruby, etc), or else to ignore the nominal locale encoding entirely and assume the use of UTF-8 (Rust, Go, Node.js, etc).

The locale coercion is done as early as possible in the interpreter startup sequence when running standalone: it takes place directly in the C-level main() function, even before calling in to the Py_Main()` library function that implements the features of the CPython interpreter CLI.

LC_CTYPE is the actual locale category that CPython relies on to drive the implicit decoding of environment variables, command line arguments, and other text values received from the operating system.

Setting LC_ALL to C.UTF-8 imposes a locale setting override on all C/C++ components in the

current process and in any subprocesses that inherit the current environment.

Setting LANG to C.UTF-8 ensures that even components that only check the LANG fallback for their locale settings will still use C.UTF-8 .

Together, these should ensure that when the locale coercion is activated, the switch to the C.UTF-8 locale will be applied consistently across the current process and any subprocesses that inherit the current environment.

PEP 540 -- Add a new UTF-8 Mode

[JLF May 6, 2018]

PEP created 5-January-2016

Abstract:

Add a new "UTF-8 Mode" to enhance Python's use of UTF-8. When UTF-8 Mode is active, Python will:

- use the utf-8 encoding, regardless of the locale currently set by the current platform, and
- change the stdin and stdout error handlers to surrogateescape.

This mode is off by default, but is automatically activated when using the "POSIX" locale.

Add the -X utf8 command line option and PYTHONUTF8 environment variable to control UTF-8 Mode.

Internals

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) :

```
    /* There are 4 forms of Unicode strings:

       - compact ascii:
         * structure = PyASCIIObject
         * (length is the length of the utf8 and wstr strings)
         * (data starts just after the structure)
         * (since ASCII is decoded from UTF-8, the utf8 string are the data)

       - compact:
         * structure = PyCompactUnicodeObject
         * (data starts just after the structure)

       - legacy string, not ready:
```

```
- legacy string, ready:

Compact strings use only one memory block (structure + characters),
whereas legacy strings use one block for the structure and one block
for characters.

Legacy strings are created by PyUnicode_FromUnicode() and
PyUnicode_FromStringAndSize(NULL, size) functions. They become ready
when PyUnicode_READY() is called.
```

Red (Rebol)

https://github.com/red/red/wiki/Unicode-Issues-with-Proposed-Resolutions]

Encoding normalization

The choice of whether to normalise Unicode text and the method of normalisation is left to the user programmer. Built-in functions will be provided to normalise Unicode strings to one of the four standard normalisations.

Case folding

Red will provide the standard locale-independent Unicode Case Folding as defined in this Unicode.org table defaulting to English as the language where necessary.
Red will enable locale sensitive case folding by providing a simple "plug-in" mechanism for other specific locale case folding.

Red will provide a separate library for full locale-independent case mapping. This library will effectively, override the standard case changing functions.

Red will enable locale sensitive case mapping by providing a simple "plug-in" mechanism in the separate library for other specific locale case mappings.

One "Character", Multiple Code Points

There is substantial extra processing in providing full "grapheme cluster" support that will not be used in the vast majority of programs written in Red. As a result, it is proposed not to change the current default behaviour from treating code points individually.

"grapheme cluster" functionality will be provided by introducing a text! datatype that will act as being "character-based". The string! datatype will be retained and continue to act as being "code-point-based". Programmers will be able to switch between the two as they need.

Reddit

http://www.reddit.com/r/programming/comments/hmtki/best_comment_about_unicode_support_in_programming/?utm_source=twitterfeed&utm_medium=twitter

Ruby

Understanding M17N

http://graysoftinc.com/character-encodings/understanding-m17n-multilingualization

Articles in Character Encodings

http://graysoftinc.com/character-encodings

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
https://redmine.ruby-lang.org/issues/2034

A runnable document about ruby 1.9's multilingualization properties.
https://github.com/candlerb/string19

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

Rust

A string is a sequence of unicode scalar values encoded as a stream of UTF-8 bytes. All strings are guaranteed to be validly-encoded UTF-8 sequences. Additionally, strings are not null-terminated

and can contain null bytes.

UTF-8 strings versus "encoded ropes"

https://mail.mozilla.org/pipermail/rust-dev/2014-May/009725.html

The begining of the discussion is focused on the "wrong" design of ruby 1.9.

In conclusion, the idea of rops in the stdlib is rejected. They prefer to convert early (I/O barrier) rather than converting later when concatenating strings of different encodings.

How to find Unicode string length in rustlang

https://mail.mozilla.org/pipermail/rust-dev/2014-May/009950.html

People expect there to be a .len(), and the only sensible .len() is byte length (because char length is not O(1) and not appropriate for use with most string-manipulation functions)

Since Rust strings are UTF-8 encoded text, it makes sense for .len() to be the number of UTF-8 code units. Which happens to be the number of bytes.

... Followed by a discussion about bytes, characters, code points, code units, graphems, ...

len(), byte_len(), units(), size(), ...

In conclusion : they keep the name "len", for the function which returns the number of bytes.

Text encoding

https://mail.mozilla.org/pipermail/rust-dev/2013-July/005015.html

https://github.com/lifthrasiir/rust-encoding

Based on WHATWG Encoding Standard :

https://encoding.spec.whatwg.org/

JLF : to investigate ? See the section WHATWG in this document.

unicode support and core

https://mail.mozilla.org/pipermail/rust-dev/2012-January/001193.html

A short discussion about to_lower/uppercase, and the risk of bad implementation for Unicode.

This is the strategy we're following, with one additional

category: tasks that satisfy all three of these points:

- Requiring some >ASCII, unicode logic
- Not-requiring any linguistic or locale-related logic
- Common-ish in routine 'language ignorant' data-processing tasks

So same conclusion than next point : don't include ICU in libcore.

unicode support and core

https://mail.mozilla.org/pipermail/rust-dev/2011-December/001140.html

A short discussion about the inclusion (or not inclusion) of ICU in licore.

The decision is to not include ICU (too big 16Mb), and to duplicate a small part of the unicode tables, for categorization of characters.

Unicode identifiers

https://mail.mozilla.org/pipermail/rust-dev/2011-February/000176.html

A short discussion about the risk of using similar font characters in symbols (i.e. variable names).

The conclusion is to add a pragma to let restrict the set of allowed characters in the symbols.

Swift

[JLF May 29, 2014]

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/StringsAndCharacters.html

JLF : Very close to what I would like to have with ooRexx.

Unicode scalar values

Behind the scenes, Swift's native String type is built from Unicode scalar values. A Unicode scalar is a unique 21-bit number for a character or modifier, such as U+0061 for LATIN SMALL LETTER A ("a"), or U+1F425 for FRONT-FACING BABY CHICK ("").

NOTE : A Unicode scalar is any Unicode code point in the range U+0000 to U+D7FF inclusive or U+E000 to U+10FFFF inclusive. Unicode scalars do not include the Unicode surrogate pair code points, which are the code points in the range U+D800 to U+DFFF inclusive.

Extended Grapheme Clusters

Every instance of Swift's Character type represents a single extended grapheme cluster. An extended grapheme cluster is a sequence of one or more Unicode scalars that (when combined) produce a single human-readable character.

Here's an example. The letter é can be represented as the single Unicode scalar é (LATIN SMALL LETTER E WITH ACUTE, or U+00E9). However, the same letter can also be represented as a pair of scalars—a standard letter e (LATIN SMALL LETTER E, or U+0065), followed by the COMBINING ACUTE ACCENT scalar (U+0301). The COMBINING ACUTE ACCENT scalar is graphically applied to the scalar that precedes it, turning an e into an é when it is rendered by a Unicode-aware text-rendering system.

Counting Characters

To retrieve a count of the Character values in a string, use the count property of the string's characters property.

Note that Swift's use of extended grapheme clusters for Character values means that string concatenation and modification may not always affect a string's character count.

For example, if you initialize a new string with the four-character word cafe, and then append a COMBINING ACUTE ACCENT (U+0301) to the end of the string, the resulting string will still have a character count of 4, with a fourth character of é, not e.

Extended grapheme clusters can be composed of one or more Unicode scalars. This means that different characters—and different representations of the same character—can require different amounts of memory to store. Because of this, characters in Swift do not each take up the same amount of memory within a string's representation. As a result, the number of characters in a string cannot be calculated without iterating through the string to determine its extended grapheme cluster

boundaries. If you are working with particularly long string values, be aware that the count(_:) function must iterate over the Unicode scalars in the entire string in order to calculate an accurate character count for that string.

The count of the characters returned by the characters property is not always the same as the length property of an NSString that contains the same characters. The length of an NSString is based on the number of 16-bit code units within the string's UTF-16 representation and not the number of Unicode extended grapheme clusters within the string.

String Indices

JLF : this is a sequential access. So no direct access, as allowed by NFG. For ooRexx, I would like a direct access, to let use subchar(n) efficiently. That will imply an array of positions. ObjectIcon has a similar approach : the ucs type maintains an index of offsets into the utf-8 string to make random access faster. The size of the index is only a few percent of the total allocation for the ucs object.

Each String value has an associated index type, String.Index, which corresponds to the positions of each Character in the string.

As mentioned above, different characters can require different amounts of memory to store, so in order to determine which Character is at a particular position, you must iterate over each Unicode scalar from the start or end of that String. For this reason, Swift strings cannot be indexed by integer values.

Use the startIndex property to access the position of the first Character of a String. The endIndex property is the position after the last character in a String. As a result, the endIndex property isn't a valid argument to a string's subscript. If the String is empty, startIndex and endIndex are equal.

You access the indices before and after a given index using the index(before:) and index(after:) methods of String. To access an index farther away from the given index, you can use the index(_:offsetBy:) method instead of calling one of these methods multiple times.

Attempting to access an index outside of a string's range or a Character at an index outside of a string's range will trigger a runtime error.

Use the indices property of the characters property to access all of the indices of individual characters in a string.

```
for index in greeting.characters.indices {
    print("\(greeting[index]) ", terminator: "")
}
// Prints "G u t e n   T a g ! "
```

NOTE

You can use the startIndex and endIndex properties and the index(before:), index(after:), and index(_:offsetBy:) methods on any type that conforms to the Collection protocol. This includes String, as shown here, as well as collection types such as Array, Dictionary, and Set.

String and Character Equality

Two String values (or two Character values) are considered equal if their extended grapheme clusters are canonically equivalent. Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they are composed from different Unicode scalars behind the scenes.

For example, LATIN SMALL LETTER E WITH ACUTE (U+00E9) is canonically equivalent to LATIN SMALL LETTER E (U+0065) followed by COMBINING ACUTE ACCENT (U+0301). Both of these extended grapheme clusters are valid ways to represent the character é, and so they

are considered to be canonically equivalent.

Conversely, LATIN CAPITAL LETTER A (U+0041, or "A"), as used in English, is not equivalent to CYRILLIC CAPITAL LETTER A (U+0410, or "А"), as used in Russian. The characters are visually similar, but do not have the same linguistic meaning.

NOTE

String and character comparisons in Swift are not locale-sensitive.

Unicode Representations of Strings

When a Unicode string is written to a text file or some other storage, the Unicode scalars in that string are encoded in one of several Unicode-defined encoding forms. Each form encodes the string in small chunks known as code units. These include the UTF-8 encoding form (which encodes a string as 8-bit code units), the UTF-16 encoding form (which encodes a string as 16-bit code units), and the UTF-32 encoding form (which encodes a string as 32-bit code units).

Swift provides several different ways to access Unicode representations of strings. You can iterate over the string with a for-in statement, to access its individual Character values as Unicode extended grapheme clusters.

Alternatively, access a String value in one of three other Unicode-compliant representations:

- A collection of UTF-8 code units (accessed with the string's utf8 property)
- A collection of UTF-16 code units (accessed with the string's utf16 property)
- A collection of 21-bit Unicode scalar values, equivalent to the string's UTF-32 encoding form (accessed with the string's unicodeScalars property)

Performance test-case

[JLF Dec 31, 2016]

https://medium.com/@tonyallevato/strings-characters-and-performance-in-swift-a-deep-dive-b7b5bde58d53#.lwduxt3nh

https://github.com/allevato/swift-performance-strings

The power of Character is that it treats code point sequences as a single "human-perceived character" and that it cleanly handles canonical equivalence, but it is not the natural representation of string elements.

Swift String values contain an instance of the _StringCore type, which is optimized to store either ASCII or UTF-16 encoded text. When Swift gives you a CharacterView, it needs to transform that underlying representation into Character values. Even though it does this lazily as you traverse the collection, if you iterate over the entire string, CharacterView is doing a lot of memory allocations and complex calculations to give you those values.

To improve the efficiency of your code when you don't need the features that Character provides, use String.unicodeScalars instead. UnicodeScalarView is a happy medium between the very high-level CharacterView and the low-level UTF8View and UTF16View. Very minor changes to your code can yield significant performance gains: 50 times faster than the version that used CharacterView.

JLF : the conclusion of Tony Allevato to improve performance is to use the UnicodeScalar view. See implementation review. This conclusion is applicable only when the grapheme view is not needed, which is the case of his test case, but NOT the general case.

Implementation review

[JLF Dec 31, 2016]

The comments are from Tony Allevato (the author of the blog about performance).

Character.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/Character.swift

The `Character` type represents a character made up of one or more Unicode scalar values, grouped by a Unicode boundary algorithm. Generally, a `Character` instance matches what the reader of a string will perceive as a single character. The number of visible characters is generally the most natural way to count the length of a string.

String.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/String.swift

StringCharacterView.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/StringCharacterView.swift

Tony Allevato:

CharacterView inherits the default IndexingIterator from Collection, so each call to next does the following:

- Uses subscript to get the current character, which constructs a new String from a slice of the buffer and then a Character from tha.
- Advances its index to the next character, which is a relatively complex algorithm in order to achieve Unicode correctness.

StringCore.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/StringCore.swift

Tony Allevato:

The core implementation of a highly-optimizable String that can store both ASCII and UTF-16, and can wrap native Swift _StringBuffer or NSString instances.

An instance variable stores a pointer to the underlying bytes that make up the string content. Another instance variable keeps track of the count of ASCII or UTF-16 code units in the string, but the most significant bit of this count is special. _StringCore calls it elementShift: if it equals 0, the buffer contains ASCII data; if it equals 1, the buffer contains UTF-16 data. In other words, adding 1 to the value of this bit gives us the number of bytes that we need to advance to get from one code unit to the next: 1 for ASCII and 2 for UTF-16.

_StringCore also ensures that internal consistency is maintained during other string operations; for example, if you append a string with UTF-16 data to one with ASCII data, the ASCII data will be widened to UTF-16.

Since _StringCore can store both kinds of text, how does it know which one to use for a given string? The compiler examines string literals and detects whether it contains only 7-bit ASCII (code units in the range 0 to 127) or if it contains Unicode, and it uses different initializers to create the String values.

StringUTF8.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/StringUTF8.swift

A view of a string's contents as a collection of UTF-8 code units.

Tony Allevato:

- A String with ASCII data accessed through UTF8View: Since ASCII is a subset of UTF-8, the data matches the way we're viewing it, so this should be fast.
- A String with UTF-16 data accessed through UTF8View: The UTF-16 data has to be transcoded to UTF-8 and the iterator must maintain state about where it currently sits within a character's UTF-8 code units, so the computational overhead would make this somewhat slower.

StringUTF16.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/StringUTF16.swift

A view of a string's contents as a collection of UTF-16 code units.

Tony Allevato:

- A String with ASCII data accessed through UTF16View: ASCII characters can be cheaply widened to 16 bit integers that are valid UTF-16 code units, so this should be fairly fast.
- A String with UTF-16 data accessed through UTF16View: The data matches the way we're viewing it, so this should be fast.

Unlike withCharacter, Swift does not let single-character string literals be used as UTF-16 code units.

There is no overload of String.append that can take a UTF-16 code unit, UTF16View is not mutable, and converting a single UTF-16 code unit to a String which could be appended is not always feasible (for example, if it is part of a surrogate pair).

StringUnicodeScalarView.swift

https://github.com/apple/swift/blob/master/stdlib/public/core/StringUnicodeScalarView.swift

A view of a string's contents as a collection of Unicode scalar values.

Tony Allevato:

UnicodeScalar conforms to ExpressibleByUnicodeScalarLiteral, so the compiler will let us create a UnicodeScalar using a string literal that contains a single scalar. In other words, we can still write case "," and have our code be readable.

In addition to the readability benefit, initializing a UnicodeScalar from a string literal is fast. UnicodeScalar is a wrapper around a 32-bit unsigned integer, and there are no string buffer allocations involved like we saw with Character.

We can't append UnicodeScalar directly to a String, but unlike UTF8View and UTF16View, UnicodeScalarView is mutable and the append(UnicodeScalar) operation can be found there.

Iterating over the UnicodeScalarView is very fast as well. The iterator delegates most of its behavior to the UTF16 codec type, which decodes the underlying ASCII or UTF-16 code units in the string buffer and produces UnicodeScalar values.

The vast majority (96.875%) of UTF-16 code units are numerically equivalent to their Unicode scalar (those in the range 0x0000–0xD7FF or 0xE000–0xFFFF). The remaining ones are the surrogate code units, and well-formed UTF-16 surrogate pairs consist of one high surrogate followed immediately by one low surrogate. The decoder never has to look more than one position ahead to convert any UTF-16 code unit or surrogate pair into a UnicodeScalar, and the conversion is simple bitwise arithmetic that does not require any additional memory to be allocated.

A downside is that each element we process no longer necessarily corresponds to a single human-recognizable character and we lose the ability to easily check for canonical equivalence.

Strings in Swift 3

https://oleb.net/blog/2016/08/swift-3-strings/

Twitter

[JLF Sep 30, 2012]

https://dev.twitter.com/docs/counting-characters

Tweet length is measured by the number of codepoints in the NFC normalized version of the text.

All Twitter attributes accept UTF-8 encoded text via the API. All other encodings must be converted to UTF-8 before sending them to Twitter in order to guarantee that the data is not corrupted.

Take the following example: \the word "café". It turns out there are two byte sequences that look exactly the same, but use a different number of bytes:

café 0x63 0x61 0x66 0xC3 0xA9 Using the "é" character, called the "composed character".

café 0x63 0x61 0x66 0x65 0xCC 0x81 Using the combining diacritical, which overlaps the "e".

Twitter does count "café" as four characters no matter which representation is sent.

Unicode normalization : http://unicode.org/reports/tr15/

Unicode

http://www.unicode.org/glossary/

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

http://www.unicode.org/reports/tr36

UTR #36: Unicode Security Considerations

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

UTF-8

http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt

http://en.wikipedia.org/wiki/UTF-8

Which is the max number of bytes for a character ? 6 bytes or 4 bytes ?

The original specification covered numbers up to 31 bits (the original limit of the Universal Character Set). In November 2003 UTF-8 was restricted by RFC 3629 to end at U+10FFFF, in order to match the constraints of the UTF-16 character encoding. This removed all 5- and 6-byte sequences, and about half of the 4-byte sequences.

So the answer is : 4 bytes...

UTF-8 Everywhere

http://utf8everywhere.org/

UCS vs UTF-8 as Internal String Encoding

http://lucumr.pocoo.org/2014/1/9/ucs-vs-utf8/

Interesting reflexion about which encoding to use internally.

The conclusion is "I'm definitely expecting more languages to take the UTF-8 route in the future".

UTF-8, a transformation format of ISO 10646

http://tools.ietf.org/html/rfc3629

In UTF-8, characters from the U+0000..U+10FFFF range (the UTF-16 accessible range) are encoded using sequences of 1 to 4 octets.

The table below summarizes the format of the different octet types. The letter x indicates bits available for encoding bits of the character number.

```
Char. number range  |        UTF-8 octet sequence
   (hexadecimal)    |              (binary)
--------------------+---------------------------------------------
0000 0000-0000 007F | 0xxxxxxx
0000 0080-0000 07FF | 110xxxxx 10xxxxxx
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

UTF-16

Should UTF-16 be considered harmful?

http://stackoverflow.com/questions/1049947/should-utf-16-be-considered-harmful

W3C

Migrating to Unicode

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

Encoding

http://www.w3.org/TR/encoding/

WHATWG (Web Hypertext Application Technology Working Group)

https://whatwg.org/

The WHATWG was founded by individuals of Apple, the Mozilla Foundation, and Opera Software in 2004, after a W3C workshop. Apple, Mozilla and Opera were becoming increasingly concerned about the W3C's direction with XHTML, lack of interest in HTML and apparent disregard for the needs of real-world authors.

The W3C publishes some forked versions of these specifications. We have requested that they stop publishing these but they have refused. They copy most of our fixes into their forks, but their forks are usually weeks to months behind. They also make intentional changes, and sometimes even unintentional changes, to their versions. We highly recommend not paying any attention to the W3C forks of WHATWG standards.

Specification for encodings :

https://encoding.spec.whatwg.org/

JLF : to investigate ?

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

WTF-8 (Wobbly Transformation Format − 8-bit)

https://simonsapin.github.io/wtf-8/

The WTF-8 encoding.

WTF-8 (Wobbly Transformation Format − 8-bit) is a superset of UTF-8 that encodes surrogate code points if they are not in a pair.

It represents, in a way compatible with UTF-8, text from systems such as JavaScript and Windows that use UTF-16 internally, but don't enforce the well-formedness invariant that surrogates must be paired.

z/OS & z/VM

Is ICU supported on z/OS and z/VM platforms?

http://marc.info/?l=icu&m=106972546516697&w=2

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/

http://98.245.80.27/tcpc/OSCON2011/gbu/gbu.html

http://www.alanwood.net/unicode/

Alan Wood's Unicode resources

[JLF Sep 30, 2012]

http://www.reddit.com/r/programming/comments/zyibm/the_importance_of_languagelevel_abstract_unicode/

Some reddit comments :

- Text is a perfect example of data that can be operated on through many different views. It is simultaneously a sequence of code units in a particular encoding, a sequence of unicode characters, a sequence of CCSes (JLF: combining character sequences), a representation of unicode characters in a particular normalization form, a sequence of graphemes, and probably more. There is a natural notion of length for each of these and none is the singular length of the text.
  There is a need for operations at many levels of abstraction. The goal of language designers should be primarily to provide mechanisms which allow these abstractions to be easily expressed; the job of the library designer is to provide the actual string operations; and the job of user is to understand how to apply the library. Trying to shield the user from the complexity of unicode is folly.
- what you want is an iterator interface. Forget about the length - it doesn't matter.
  for c in chars(str)
  for g in glyphs(str)
  for b in bytes(str)

[JLF Apr 17, 2017]

http://illegalargumentexception.blogspot.co.uk/2009/04/i18n-unicode-at-windows-command-prompt.html#charsets_1252

I18N: Unicode at the Windows command prompt (C++; .Net; Java)

http://illegalargumentexception.blogspot.co.uk/2010/04/i18n-comparing-character-encoding-in-c.html

I18N: comparing character encoding in C, C#, Java, Python and Ruby

https://github.com/Ed-von-Schleck/shoco

shoco: a fast compressor for short strings

http://srfi.schemers.org/srfi-75/mail-archive/msg00270.html

freshman-level Boyer-Moore fast string search

https://www.codeproject.com/Articles/10661/Efficient-Boyer-Moore-Search-in-Unicode-Strings

https://www.codeproject.com/Articles/14637/UTF-With-C-in-a-Portable-Way

UTF-8 With C++ in a Portable Way

https://www.codeproject.com/Articles/17201/Detect-Encoding-for-In-and-Outgoing-Text

Detect Encoding for In- and Outgoing Text

https://www.codeproject.com/Articles/774909/Fun-with-Unicode

Fun with Unicode

https://www.codeproject.com/Articles/16520/FormatString-Smart-String-Formatting

FormatString - Smart String Formatting

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.

[JLF May 14, 2015] JLF : see my notes later. I don't see the problem... Just assume that these functions work on bytes, whatever the encoding. Even the length function must be seen as returning the number of bytes.
About xrange : the ooRexx doc says *The default value for start is "00"x, and the default value for end is "FF"x*. That's clear : ~~ASCII~~ byte range, and nothing else.

...

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. You can have an UTF-8 string showing one character (one grapheme) made of several bytes. The behavior of ooRexx is correct : returns all the bytes. For me, the problem is more how you isolated the character... The ooRexx functions work on bytes, so isolating a character encoded with several bytes implies to know the encoding. That should be handled by a class other than String. For legacy compatibility, String should remain one-byte encoded (i.e. binary 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'
```
JLF : no xrange in netrexx ? Called sequence ? Ok, sounds good... In ooRexx, xrange is limited to the ASCII encoding (see the doc : defaults from "00x" to "FF"x).

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).

Mailing list

http://ibm-netrexx.215625.n3.nabble.com/Re-Welcome-to-the-quot-Ibm-netrexx-quot-mailing-list-td680827.html#a683436

Mike Cowlishaw

In reply to this post by Chip Davis
Chip wrote:

> I was going to suggest:
>
> s = "a0987654321sadf09253490-3t4l'dkfgdi09531=5798"
> Say s.translate('1234567890','1234567890'||xrange)
>
> only to find that xrange() is conspicuous by its absence.
> I'm not sure why.
>
> Mike, can you elaborate?

Character strings in NetRexx/Java are Unicode (the subset that can be UTF-8-encoded), whereas xrange in 'classic' Rexx produces a series of bytes, which could really only be represented as a byte array. However, all the "BIFs" work on character strings (of type Rexx). Indeed, the NetRexx language as such does not require that binary data (such as bytes) exist, although it admits that implementations might support such antique concepts :-).

JLF : netrexx provides 'sequence' which does the same as 'xrange', but not limited to ASCII.

For the same reason x2c (for example) can only produce a single character.

Translate should be fine.

Mike

Mike Cowlishaw

In reply to this post by Chip Davis

> only to find that xrange() is conspicuous by its absence.
> I'm not sure why.
>
> Mike, can you elaborate?

Forgot to add: also, the default call to xrange in Rexx returns 'all possible character encodings'; that would be a very long string in NetRexx.

Mike

Patric Bechtel

Hi Mike,

sorry, wasn't my intention telling that translate does not work. It's just the wrong tool for the given usecase. It's good if you want to translate something into something else, but it's just plain condemned to fail in a case, where it should replace "everything but" in a unicode case.

Mike Cowlishaw

In reply to this post by Chip Davis
> I guess it's time to acquire a whole new set of character
> string manipulation idioms.

Not necessary .. almost all the Rexx BIFs were/are directly preserved in NetRexx. The only ones that needed changes are the '2c' and 'c2' ones (and xrange) -- that assumed one character = one byte. Maybe five in total .. everything else just works :-)

Mike

Mike Cowlishaw

> Hi there, may I add that the www.Rexx2Nrx.com run-time
> package contains the functions c2x and x2c with STRINGS as
> the argument as well....

So what does it do when (for example) c2x has characters in the input string such as \u8212 (emdash)? If all characters go to 16 bits, then it's not compatible with classic Rexx. If they are decapitated to 8 bits, then c2x and x2c are not reversible. And then there are 32-bit Unicodes (which Java did not support when I did NetRexx, but maybe it does now).

JLF : the risk of decapitation is not here... In ooRexx, the decapitation will happen when using the functions left, right, ... which all work on bytes. See my comment for c2d().

Mike

rickmcguire

Indeed. Trying to maintain compatible behavior with this was a major impediment to any plans for making ooRexx change to Unicode-based strings. I was constantly seeing presentations showing how to accomplish different things that would just fail once characters were no longer 8-bits long. Bob's translate() example happens to be one of those.
Rick

JLF : Just don't change the behavior of String. This class work on bytes, period. New classes will be needed to provide more abstract services. Not only at codepoint level, but also at grapheme level, word level, paragraph level, ...

JLF 11/09/2016 : Several classic Rexx scripts in RosettaCode contain UTF-8 characters. Ex: in Aliquot-sequence-classifications, there is

```
say center('numbers from ' low " to " high, 79, "═")
```

where "═" is the Unicode character BOX DRAWINGS DOUBLE HORIZONTAL, encoded E2 95 90 in UTF-8. ooRexx and Regina complains:

Error 40.23: CENTER argument 3 must be a single character; found "═"

I would like an intelligent support of UTF-8 here, without modifying the Rexx script. The length in grapheme is 1, the interpreter should not complain, even if the length in bytes is <> 1.

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).

[JLF Mar 15, 2013] Warning : this wide-char version is derived from an older version of ooDialog (april 2010), and is no longer in sync with ooDialog 4.2.0 (the byte-char version).
I don't plan to redo this work.

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.

[JLF Mar 15, 2013] Parrot has simplified the internal representation : Charsets have been merged into encodings. Now they need only the encoding.
The list of supported encodings is :
ascii
iso-8859-1
binary
utf8
utf16
ucs2
ucs4

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. [JLF Mar 15, 2013] Better: UTF-8.

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 [JLF Mar 15, 2013] Better: UTF-8.

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 Rexx 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
```