UNICODE

Stephen Schaub

Facts of Life

- FOL #1: Computers store and transmit data in units of bytes
- FOL #2: The world's languages require more than 256 characters
  - Single-byte encodings like ASCII map individual bytes to characters
  - All single-byte encodings pretend that FOL #2 doesn't exist

A Brief History of Encoding

- 1 byte per character schemes
  - ASCII
  - Code page systems
- 2 bytes per character schemes
  - Asian languages
- Unicode
  - 1-4 bytes per character

Unicode

- Maps characters to code points
  - A **code point** is a unique number that signifies a particular character
  - Current count: Over 137,000 code points / characters
- Code points 0-127 correspond to ASCII code

Unicode Planes

- Code points organized into 17 planes
  - Each plane can represent up to 65,535 code points
  - Plane 0: Basic multilingual plane (BMP)
    - Characters for almost all modern languages; several symbols
  - 16 supplementary "astral" planes
    - Historic languages
    - Music notation
    - Emoji
- Room for over 1 million code points
  - Most will likely never be assigned

Unicode Encodings

- Remember FOL #1? We need a way to represent code points using bytes.
- Various encodings possible
  - UTF-8
  - UTF-16
  - UTF-32
- UTF-32 can represent any possible code point in a single 4-byte value
  - Rarely used in practice (too inefficient)
- Both UTF-8 (1 byte values) and UTF-16 (2 byte values) are **variable-length** encoding systems
  - UTF-8 requires 1-4 bytes to represent a given code point
  - UTF-16 requires 2 or 4 bytes to represent a given code point

UTF-8

- Most widely used encoding of Unicode
- Requires 1-4 bytes to represent a given code point

48	69	e2	84	99	c6	b4	e2	98	82	e2	84	8c	c3	b8	e1	bc	a4
H	i	ℙ			ƴ		☂			ℌ			ø		ἤ		

HTML Page Encoding

- Content-type header can specify encoding
  - Content-Type: text/html; charset=utf-8
- Document charset meta tag can specify encoding

```
<html>
<head>
<meta charset="utf-8">
</head>
…
```

Practical Problems

- Variable-length encodings are memory-efficient, but not friendly for random access
  - Consider a string API that allows you to index a Unicode string:

    -
      ```
      for (var i = 0; i < str.length; ++i)
        console.log(str[i])
      ```
  - What are the options for implementing this behavior?

Practical Problems

- Some characters have multiple code point representations
  - Example: An accented e (é)
    - é (U+00E9)

      or
    - e (U+0065) + accent (U+0301)
  - Example: The sequence fi
    - ﬁ (U+FB01)

      or
    - f (U+0066) + i (U+0069)
- Comparing Unicode strings for equality can be tricky

Case Folding

- Problem: Need to compare two strings to see if they contain the same letters, ignoring capitalization
  - With simple ASCII, case-insensitive comparisons are straightforward
  - Convert all letters in a string to the same capitalization (“case fold”) and then compare for equality

Case Folding

- With Unicode, the problem of comparing strings is more complex
- Three distinct issues:
  - Ignoring different Unicode representations for the same characters
    - Example: Compare “é” (U+00E9) to “é” (U+0065 U+0301)
  - Ignoring capitalization
    - Example: Compare “MASSE” to “Maße”
  - Ignoring accents, diacriticals, and other symbols
    - Example: Compare “Saens” to “Saëns”
- See https://www.w3.org/International/wiki/Case_folding

Handling Different Unicode Representations

- Unicode defines normalization forms and algorithms that convert two strings with different representations to the same representation

- Further reading:
  - See https://en.wikipedia.org/wiki/Unicode_equivalence

Handling Capitalization and Diacritical Differences

- Unicode defines algorithms for caseless matching
  - Libraries are available for various languages that implement this algorithm
- Other algorithms are available for removing accents

Case Studies

Searching Unicode Text

- BJU Digital Music Project
- Database contains composition titles and composer names like
  - Camille Saint-Saëns
- Problem: Searches need to match Saens == Saëns
- Solution:
  - Remove diacritical marks
  - Compare using case folding

Printing Source Code

- Code Listings Utility
- Problem: Print source code in unknown encodings
- Solution: Node detconv
  - Uses port of https://github.com/chardet/chardet to detect encoding
  - Uses https://github.com/ashtuchkin/iconv-lite to convert to ascii

JavaScript and Unicode

JavaScript and Unicode

- JavaScript represents strings internally using UCS-2
  - UCS-2 is a limited version of UTF-16 that handles only Plane 0 (BMP)
- Strings containing only Unicode values in the BMP (U+0000 to U+FFFF) often work well
  - Each of these can be represented using a single 16-bit value

Accented Characters

- é can be stored using one or two code points:
  - const s1 = "\u00e9"; // é
  - const s2 = "\u0065\u0301"; // é
- Both strings represent a single character
- JavaScript's length property counts code points, not characters
  - s1.length == 1
  - s2.length == 2
- Guess what you get when you index s2[0]? (Browser demo)

More JavaScript

- Consider emoji's 😀
  - Not in the BMP
  - Require a double-length UTF-16 value
  - JavaScript's string API exposes the underlying 16-bit representation in undesirable ways

Accurate indexing and character counts

- Unfortunately no good solutions exist at present in native JavaScript
- Node.js: Punycode library can help
  - Punycode represents Unicode in ASCII:
  - https://en.wikipedia.org/wiki/Punycode
- See https://dmitripavlutin.com/what-every-javascript-developer-should-know-about-unicode/ for more suggestions

Comparing Strings

- Use the .normalize() method to convert two strings to a canonical representation that can be compared
  - const s1 = '\u00E9' // é
  - const s3 = 'e\u0301' // é
  - s1 !== s3
  - s1.normalize() === s2.normalize()
- Note that normalization preserves capitalization
  - The comparison is case sensitive
- No built-in way to do reliable case insensitive comparisons in JavaScript exists in 2020

Converting Bytes to Unicode

- Node.js: Byte data from files / sockets arrives as Buffer
- If data is textual, must convert Buffer to string
  - Specify encoding
  - let str = buf.toString('utf8')
- How do you know encoding?
  - Must be told
  - In general, not possible to infer

You Must Be Told the Encoding

- Cannot infer from a string of bytes; can only guess

	48	69	e2	84	99	c6	b4	e2	98	82	e2	84	8c	c3	b8	e1	bc	a4
utf-8	H	i	ℙ			ƴ		☂			ℌ			ø		ἤ		
iso8859-1	H	i	â	„	™	Æ	´	â	˜	‚	â	„	Œ	Ã	¸	á	¼	¤
utf-16-le	楈		蓢		욙		⍰		芘		蓢		쎌		⍰		꒼	
utf-16-be	䡩		⍰		駆		듢		颂		⍰		賃		룡		벤	
shift-jis	H	i	邃		卮		ｴ	笞		やc		ъ		ﾃ	ｸ	眈		､

Python and Unicode

Python and Unicode

- Python 3 has two sequence types that can hold textual info
  - bytes (unencoded byte values)
  - str (UTF-8 encoded Unicode)

```
>>> my_string = "Hi \u2119\u01b4\u2602\u210c\xf8\u1f24"
>>> type(my_string) <class 'str'>
>>> my_bytes = b"Hello World"
>>> type(my_bytes) <class 'bytes'>
```

Normalization and Case Folding

- Normalize:
  - import unicodedata
  - if unicodedata.normalize('NFC', s1) == unicodedata.normalize('NFC', s2)
- Case insensitive compare:
  - Basic strings: s1.lower() == s2.lower()
  - Better: s1.casefold() == s2.casefold()
- Combining the two:
  - See https://stackoverflow.com/a/40551443

Pain Relief

Tip #1: Unicode Sandwich

- Bytes on the outside, Unicode on the inside
- Encode/decode at the edges
  - Receive binary data
  - Decode immediately to Unicode
  - Process as Unicode text
  - Encode as late as possible
  - Send binary data

Tip #2: Know What You Have

- Have a Buffer containing textual data?
  - Convert to string, specifying encoding
- Have a string and need a buffer?
  - Convert to buffer, specifying encoding

Tip #3: Avoid guessing the encoding

- Penalty for guessing wrong:
  - Some characters may fail to decode
  - The string may decode successfully, but produce wrong characters

Pro Tip #4: Test with unicode

- ȧƈƈḗɳŧḗḓ ŧḗẋŧ ƒǿř ŧḗşŧīɳɠ
- ℛℯα∂αbℓℯ bʊ☂ η😊т Ⱥ$☾ℐℐ
- ¡ooɟ ןnɟəsn sı uʍop-əpısdn

References

- Ned Batchelder, "Pragmatic Unicode." https://nedbatchelder.com/text/unipain.html
- "What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text" http://kunststube.net/encoding/
- Python 3 Unicode https://docs.python.org/3/howto/unicode.html