Abstract

Although character strings are a core datatype in many programming languages and may appear to have a straight-forward design, their design and implementation has changed significantly over the past 50 years. Language designers have creatively responded to changes in business requirements around performance, security, and internationalization to design a variety of effective solutions. This article charts the engineering design decisions in popular, multi-domain languages and the trade-offs made between representation, functionality, and multi-lingual support.

Fifty years of strings: Language design and the string datatype

Jeffrey Starr

February 6, 2022

1 INTRODUCTION

In a programming language, a string is a sequence of characters that often represents some message to the user or contains input such as a name or the contents of a file. Common operations on strings include comparison and collation, extraction of substrings (tokenization), pattern matching, and data clean up such as trimming whitespace. In the 1960s, language designers started to add first-class string datatypes to their languages as business computing and interactive use cases expanded. Although programmers could use numerical arrays as strings (as was the practice in Fortran at the time), character-based strings simplified programmer's jobs through re-use of library functions and reduction of errors caused by accidental intermixing of numeric and character based arrays. Today, a language implementation may feature hundreds of built-in functions for string manipulation, presentation, and localization.

Language designers have flexibility around the in-memory representation and storage of the string, encoding of characters, breadth of built-in functionality, and facilities for supporting a multi-lingual environment. Their choices impact performance, security, and usability. Over the past 50 years, designers have explored many options and although there is convergence in a number of areas, there is still room for innovation.

1.1 WHAT IS A CHARACTER?

A string is a sequence of characters. However, a character is both a physical and semantic concept. Physically, characters are stored as one or more integral numbers with an implicit or explicit character set. A character set is a mapping of numbers to a semantic meaning. Semantically, a character may map to a graphical symbol or to a grapheme. A symbol has a visual appearance, but a grapheme is the smallest functional unit in a language and may not necessarily have a single visual representation. The semantics of a character depend on the character set. (Characters may also have only semantic meaning to a computer, such as the line-feed control character in ASCII. Some languages, like C, use the `char` data type to also store random bytes, which has implications on memory representation as noted below.)

To illustrate the distinction, the abstract character "return" has a single grapheme (Unicode U+8FD4), but the visual representation is different in Chinese, Japanese, and Korean (see Figure 1). A character set based on graphemes would have a single entry for return while one based on symbols would have at least five entries. Unicode adopted the grapheme approach but the complexities of that approach have left indelible traces in programming languages that sought early Unicode interoperability (see Section 5).

Figure 1: Differences for the same Unicode character (U+8FD4) in regional versions of Source Han Sans [10]

Table 1: String Representations in Popular Languages

	COBOL	Pascal (UCSD)	C (K&R)	Smalltalk	Objective-C	Perl 1.0	Python 1.0	Ada 95	Java 1.0	C++ 98	C# 1.0	Go	Swift 1.0	Rust 1.0
Release Year	1968	1978	1978	1980	1986	1987	1994	1995	1996	1998	2002	2012	2014	2015
Representation	FL	LP	NT	RE	NT	RE	RE	RE	RE	RE	RE	RE	RE	RE
Character Type	B	B	B	B	B	B	B	BW	U16e	BW	U16	B/U8	U16	U8
Immutable?	No	No	No	No	No	No	Yes	No	Yes	No	Yes	Yes	No	No
Reference	[20]	[25]	[17]	[12]	[9]	[29]	[28]	[4]	[13]	[26]	[15]	[22]	[5]	[18]

2 TIMELINE AND MAJOR DESIGN CHOICES

Table 1 lists languages with their release year, in-memory representation of strings, character types, and whether strings are immutable. For inclusion in this timeline, we selected languages that achieved popularity, were intended for industrial use across multiple problem domains, and where the designer had latitude in design decisions. This list excludes many languages that run on the Java Virtual Machine or Common Language Runtime as those languages would inherit that environment's standard string representation. Spreadsheets were also excluded because their design was not intended to be multi-domain. The date for each language is based on their first major public release or when there was a significant change in their handling of strings. Programmers could acquire libraries that expanded string functionality or implemented a different approach from the underlying language, but our focus is on choices made by language designers. For example, the C++ standard used C strings until the 98 version introduced `std::string`, but there were many common libraries that added alternative string support to C++.

Representations:

FL fixed length array (whitespace padded)

LP array of characters prefixed with length (length is inline)

NT array of characters terminated by NUL or other sentinel value

RE record (or richer) encapsulation of the in-memory representation

Designers that chose a record for the in-memory representation often used the opportunity to store additional data such as pre-computed hashes of the content or specialize the representation such that short strings will take less space. For interoperability, designers may also choose to include a NUL byte within the array, or store a length-prefix byte, even if the language does not expose this fact.

Character types:

B Byte (Older languages may support 6-bit bytes)

BW Byte (Octet) or Wide Character

U16e 16-bit Unicode, prior to creation of surrogate pairs (UCS-2)

U16 16-bit Unicode (UTF-16)

U8 8-bit Unicode (UTF-8)

If a designer decides that strings will follow a specific character encoding, they need to decide how strictly that guarantee will be met. Prior to the mid-1990s, designers allowed strings to hold arbitrary bytes. An individual program could enforce that a string held bytes that were valid per some encoding and character set, but that was not a language concern. Since neither ISO 646 nor ISO 8859 character sets, which include ASCII, prohibited any specific byte values or sequences, conformance was a non-issue. With the adoption of Unicode, though, which did include invalid sequences, language designers had a choice whether a string was

always valid or whether an initialized string's operations could be in an error state. In Go's case, the string type can store arbitrary bytes but the bytes are processed as if they are encoded with UTF-8 [22] which may lead to runtime errors. In contrast, Rust enforces that strings are always valid so once a string is constructed, there are no runtime errors [3]. Since there are performance costs for validating UTF-8 encoding, and some use cases require working with potentially bad or corrupt data, this is an implementation trade-off between supporting some use cases more directly and at higher performance versus type safety and avoidance of run-time errors.

3 IN-MEMORY REPRESENTATION

All the languages in Table 1 leverage an array to store a string (although that fact may not be exposed and, in Swift's case, there are multiple potential backing stores for a string), so the in-memory representation of strings is a reflection of how the language represents arrays. The array may be storing bytes or wide characters, but the storage is contiguous.

Not all languages choose to use an array as the backing store. Historically, some languages packed characters within a word (e.g. 36-bit architectures included 6, 7, and 9 bit packing schemes) but with byte-addressing and richer type systems designers have stopped using this approach. More recently, Haskell's designers implemented the core `String` class with a linked list of Unicode code points, but Haskell practitioners tend to use alternative packed-array based representations for better performance [1] as the linked list representation requires several words of memory per character and most operations on a list are more expensive than an array.

One of the earliest representations of strings in a higher-level language was the Hollerith datum or constant, as implemented in Fortran 66. Syntactically, the punch cards featured a nH followed by n characters. The remaining n characters were stored as an array of integers, padded if necessary to the given length, as Fortran 66 did not have a character type. (Fortran 77 normalized the syntax with quoted strings.) COBOL 68 similarly represented strings as padded fixed-length sequences, but featured alphanumeric data types — characters and numbers were distinct.

Pascal (as initially defined by Jensen and Wirth) and Ada 83 represented strings as fixed-length arrays. Both of these languages featured a strong type system and chose to incorporate the length of the array within the type definition. Thus, an array of 64 characters was a distinct data type than an array of 128 characters. Lacking a way to reference a generic array of character, programmers were restricted from writing generic string functions. To work around these limitations, implementations, such as the influential UCSD version, added non-standard functionality. UCSD used length-prefix arrays, which proved popular and saw adoption in multiple Pascal implementations. Later versions of Pascal and Ada added standardized ways to interact with variable length strings. More recently, Go also chose to include the length of the array in its type definition. However, Go's use of slices, or views into the array, permit the creation of generic functions.

Length-prefix arrays are arrays of characters where the first character's ordinal value is interpreted as the length of the string. So-called "Pascal Strings", these could range up to 255 characters (the maximum size of an unsigned byte). The term pascal string is a misnomer both because it was popularized by the UCSD implementation of Pascal, not by the Pascal language standard itself, and the fact that the technique predated the language and implementation by at least a decade. In 1968, the Multics PL/I compiler used the length-prefix technique to store the length of varying length strings [11]. The PL/I language used the length for arrays bounds checking.

A distinction between length-prefix and Hollerith constants is that the length-prefix is interpreted as a machine integer while Hollerith constants were interpreted as human-input digits. Net strings are a modern version of Hollerith constants. Since multiple digits are allowed for Hollerith constants (and net strings), much larger lengths can be modeled but at a greater cost in storage of the length.

While the length prefix representation is efficient in storage requiring just a single byte, maintains locality of definition, and is still widely used in "on the wire" protocols (see appendix in [19]), it has limitations. The straight-forward encoding of the length as a single byte limited strings to a maximum length of 255 characters. (Some implementations with wide characters allowed lengths up to 65,535. A more complex representation could allow for greater range by using multiple bytes, but although the approach was discussed, it was not widely adopted.) Furthermore, because the length was inline to the array, programmers had to be careful to

not invalidate the length and start processing after the first index.

In the 1980s, implementations started to converge on NUL terminated strings. A NUL terminated string is an array of characters ended by the NUL or zero value character. (Languages could use other symbols for termination, such as B's sequence of "*e" [24] but NUL was convenient and widely available [23].) In contrast to length-prefix arrays, this representation allows for arbitrarily long strings and removes the need for special logic around the start of the array. Pointers can also index into the array to model a changing start position without requiring copying of the data, thus rendering many parsing and scanning operations cheaper. However, computing the length of a string requires a linear search for the terminator symbol whereas the length-prefix representation has constant time lookup.

Dennis Ritchie defended the design choice in C in his History of Programming Language presentation with:

> Aside from one special rule about initialization by string literals, the semantics of strings are fully subsumed by more general rules governing all arrays, and as a result the language is simpler to describe and to translate than one incorporating the string as a unique data type. Some costs accrue from its approach: certain string operations are more expensive than in other designs because application code or a library routine may occasionally search for the end of a string, because few built-in operations are available, and because the burden of storage management for strings falls more heavily on the user. Nevertheless, C's approach to strings works well. [24]

Although NUL terminated strings are associated with C, C did not originate them. A likely influence on C's use of NUL terminated strings was the existence of the ASCIZ (ASCII with a zero suffix byte) assembly macro instruction introduced for the PDP-10 (as part of MACRO-10) and carried forward for the PDP-11. This macro created NUL terminated strings. Architecturally, NUL terminated operates similar to reading from a tape where the quantity of data is unknown but there is an terminating symbol at the end.

The inability to represent the NUL byte embedded within a string was seen as an acceptable restriction for some languages (e.g. in C, the programmer could leverage the underlying array and avoid NUL-expecting library functions). Python, which did not differentiate between byte strings and strings in its initial type system, stored both an explicit length in the `objstring` record and terminated the internal character array with a NUL in order to be both interoperable as well as handle arbitrary byte sequences. The fact that the strings were NUL terminated is not exposed within the language itself, but is a useful property for Python modules that leverage the cPython's internal API.

Neither representation above supports encoding the *capacity* of the underlying buffer or array (in cases where the current length of the string does not match the capacity). If the underlying array's capacity cannot be checked (like in C for dynamically allocated arrays), programs in the language become vulnerable to buffer overflows and buffer over-reads. These are vulnerabilities where a program can access a memory location outside the bounds of the array, which can lead to intentional or unintentional memory reads or memory corruption.

Array bounds checking was initially seen as a way of protecting a programmer from their own mistakes, but the Morris worm demonstrated publicly the information security implications of unchecked access [8]. The need for better security led to a resurgence of "checked" languages that included features such as bounds checking.

By storing information about a string "out-of-band" from the underlying buffer, languages can support flexible use cases efficiently and provide better security. For example, in the 1972 version of SNOBOL [14], strings were represented by a "qualifier", a record with $< T, F, V, O, L >$ fields. The T and F fields stored the data type and metadata about the type, the V was a memory pointer to an array of characters, O was an offset within that array, and L was the length of the access. Access to a string was always made through a qualifier. This representation allowed slices of strings to be represented without requiring any new copying or allocation. This was an important capability due to SNOBOL's heavy use of pattern matching. In comparison, neither length-prefix or null-terminated in-band representations allow a slice to have the same ergonomics as the base string.

For languages where arrays can be queried for their capacity, this out-of-band mechanism is the array itself. For instance, the Java virtual machine has an instruction to return the length of an array. The storage of array lengths is abstracted away. With the capacity persisted, languages can check array accesses and provide safety guarantees. (Bounds checking can sometimes be achieved statically at compile time rather

than run-time.) However, because the in-memory representation is encapsulated rather than being exposed in a known addressable manner, interoperability between processes requires more complex solutions than just `mmap`'ing shared memory and exchanging addresses. Since security requirements have pushed for greater isolation between processes and more safety guarantees, this is another trade-off language designers make between performance, interoperability, and security.

The debate between safety, necessary use cases, and the cost of array bounds checking rages on. As a recent example of a systems language, Rust implements bounds-checking on arrays but includes escape hatches to allow for arbitrary and unchecked memory access, thus demonstrating the range of options available to designers.

4 BUILT-IN FUNCTIONALITY

The ergonomics for strings is driven by their ease-of-use and included functionality. As soon as languages started to support strings as first-class values, they included standard functionality such as string length computation, access to individual characters, and concatenation.

The 1965 manual for PL/I running on the IBM360 lists the following generic string functions: substring, string index (contains), length, high and low (uppercase and lowercase conversion), concatenation, comparison/collating, repeat (repeated self-concatenation), and several functions for conversions to and fro character string and bit strings. This is a significant list of functionality; for comparison, COBOL in 1968 only included concatenation, a method to count the number of occurrences of a substring, and a tokenization routine. PL/I is often criticized as a very large language (for its time) and most languages that followed in the next decade provided a more limited set of functions. In fact, the first release of Ada in 1983 included no string functions at all (the language design of arrays rendered generic string functions infeasible). Objective-C in 1986 had similar functionality on release as C from nearly a decade earlier, supporting comparison, concatenation, length, and replace functions.

From the late 1980s on, string functionality starts to standardize, with functions for comparison, concatenation, contains, length, replace, substring or slices, tokenization, and trimming of whitespace appearing in all the survey languages. In contrast to Ada 83, Ada 95 includes everything from the previous list except tokenization.

Independent of other design choices, a language designer may decide to use immutable strings. An immutable string is one where operations do not change the underlying data, but only change a view of the string or create a copy. Immutability simplifies performance improvements such as caching, re-use of strings (in cases where duplicates are likely), and concurrent access to the data. Immutability carries a trade-off that certain operations, like replacement, can be more expensive in time and memory.

SNOBOL included immutable strings in the 1960s (in this case, to save memory by allowing many partial views of a string) but their adoption expanded in the 1990s. Both Python and its predecessor ABC had immutable strings [27]. In Java's case, immutability was borne from security requirements — if a string (e.g. a file path) could change during a virtual machine authorization check, it could be used to defeat the check [2]. As arrays are not immutable, this is an indication that language designers saw value in distinguishing use cases of strings versus arrays of characters.

Although immutable strings have been adopted in many languages, there is not an engineering consensus on their use. The two most recent language examples in the list, Swift and Rust, use mutable strings (although they have various immutable views on a string). Both of these languages instead support a "mixed-mode" where mutability can be used if necessary, but mutability is not the default access pattern.

5 MULTILINGUAL SUPPORT

By the 1970s, computer systems were converging on a 8-bit byte or octet. (Trigraph support in C, for support of 6-bit source character sets, is a relic from this era [7].) With different hardware manufacturers implementing their own character sets, sharing data between systems was difficult. The U.S. government put their weight behind the ASCII character set in order to establish some baseline of interoperability. Although ASCII was sufficient for most U.S. applications, it required modification to work for even other English-speaking countries. To support their own languages or add characters for specialized applications

like box drawing, governments and manufacturers implemented new code pages. These alternative character sets tended to be similar to ASCII and simply changed a few characters (e.g. ISO/IEC-646 used ASCII as a baseline but created space for "national characters" that could represent certain letters or currency symbols). (Meanwhile, IBM systems stuck to EBCDIC, which required the creation of parallel national code pages.) Character sets also started to use the full 8-bit space such as ISO-8859. The seven bit space of ISO/IEC-8859 was sufficient for locales using the Latin script, but the space was too small for languages such as Chinese, Japanese, and Korean.

To support the thousands of characters required for Chinese, Japanese, and Korean, designers followed one of two implementation strategies: wide characters or multi-byte characters. (The terminology is standard but confusing, since both encodings require multiple bytes. Alternatively, the two strategies could be called fixed-length and variable-length.) The wide character strategy expanded all characters from an 8-bit to a 16-bit value while the multi-byte strategy used shift characters to use one, two, or potentially more bytes per character. The advantage of wide characters is simplicity — computing the octet length is straightforward and access to a character is a simple array look-up — while the advantage of multi-byte is more efficient use of space when the range of values are not used uniformly. If the consuming program used the wrong character set, a wide character encoding would include the NUL byte, leading to early truncation errors for programs using NUL terminating strings.

Although code pages allowed tailoring of computer systems for specific locales, there were problems with interoperability. First, a file reader had to know a priori the code page for a file as there was no way to identify the code page being used within the encoding. Second, conversions between code pages were lossy because characters might be missing between the source and destination. Third, each major vendor maintained their own database of code pages and those databases diverged, even with multi-vendor attempts at cooperation. Thus, files may not even be portable between computer systems within the same locale. This was a major tax on internationalization of software and the increasing need to share digital data globally.

In 1988, Joe Becker started discussing Unicode publicly, first at Uniforum in Dallas, Texas and he later released the Unicode '88 proposal at ISO WG2 [30]. "Unique, universal, and uniform", the new character set was proposed as one that would "encompass the characters of all the world's living languages" [6]. Becker proposed using a fixed width 16-bit wide character encoding, and argued that the simplicity of the representation was a better tradeoff than the storage savings from a multi-byte encoding. As there was an open debate on whether 16-bits would be sufficient for CJK languages as well as all other modern scripts, Becker made two arguments:

1. A character is not a glyph. Becker argued that a grapheme approach would fit within the space budget. Nations could map the semantic meanings to their language's glyphs, thus handling at the graphical symbol resolution at the typeface level.

2. Modern use only. Since the scope of the project is to only handle widely used, in-use languages, many characters can be ignored.

Unicode starts to win against competing standards and the two-volume definition of Unicode 1.0 is published in 1992. Version 1.0 describes a single wide character encoding (where each value corresponds to a single code point). To handle byte order ambiguity, the standard includes byte order markers. Version 2.0, published in 1996, incorporates the UTF-7 and UTF-8 encodings, and a surrogate mechanism to encode characters outside the 16-bit range. The surrogate mechanism is meant to "allow representation of rare characters in future extensions of the Unicode standard". At this point, there are no characters defined in the surrogate area. The addition of surrogates effectively switches the wide character encoding to a multi-byte encoding, although implementations made before the change (or implementations that deliberately ignore everything outside Unicode's basic multilingual plane) remain wide character based. The former encoding becomes the UCS-2 encoding while the UTF-16 encoding includes support for surrogates.

The use of surrogates, but without assigning any code points to that plane, continued through version 3.0 published in 1999. Version 3.1 in 2001 added 42,711 characters to the "CJK Unified Ideographs Extension B" surrogate plane. This was approximately 2/3 of the original 16-bit space of 65,536 and double the originally allocated amount of 20,940 characters for Chinese, Japanese, and Korean ideographs. Since the original 16-bit design objective was no longer tenable, version 4.0 in 2003 carried a significant design change. In the previous two versions, the very first design principle (of 10) was "Sixteen-bit character codes" with the scope

of Unicode defined as only covering modern use characters. Version 4 changed this design principle (the only one to change) to "Universality" and expanded the scope of Unicode to include historic writing systems.

These changes rendered the simple UCS-2 encoding untenable if a program wanted to support Unicode and target the world's markets. Languages that were designed in the interim required some redesign. For example, Ada 95 introduced wide characters to the language and fixed the width at 16-bits; this necessitated the addition of "wide wide" characters in the 2003 version of the language. Java kept 16-bits for individual characters, but expanded the `Character` functionality to accept 32-bits ints as code points and implemented surrogate support. C# followed a similar strategy of using UTF-16 and surrogate pairs. For C and C++, `wchar_t`'s width was always implementation-defined (e.g. Windows used 16 bits, GNU libc used 32 bits), so this avoided changes to the language. However, the ambiguity of an implemented-defined width led to `char8_t`, `char16_t`, and `char32_t` being added to the C++ standard.

The UTF-8 encoding was designed in 1992 by Rob Pike and Ken Thompson with the goals [21]:

> 1) Compatibility with historical file systems:
> Historical file systems disallow the null byte and the ASCII slash character as a part of the file name.
> 2) Compatibility with existing programs:
> The existing model for multibyte processing is that ASCII does not occur anywhere in a multibyte encoding. There should be no ASCII code values for any part of a transformation format representation of a character that was not in the ASCII character set in the UCS representation of the character.
> 3) Ease of conversion from/to UCS.
> 4) The first byte should indicate the number of bytes to follow in a multibyte sequence.
> 5) The transformation format should not be extravagant in terms of number of bytes used for encoding.
> 6) It should be possible to find the start of a character efficiently starting from an arbitrary location in a byte stream.

The UTF-8 encoding was first used in the Plan 9 operating system and saw broad adoption in Unix systems early on. By the 2010s, the UTF-8 encoding has won both in terms of serving as the majority encoding on the internet, and being used internally within programming languages. Go and Rust were launched with UTF-8 support, and although Swift initially used UTF-16 for backwards compatibility with Objective-C, switched to UTF-8 in 2019 [16].

5.1 OTHER ATTEMPTS

Unicode was not the only multilingual character set being developed in the 1980s. The ISO organization sought to create a Universal Coded Character Set (UCS) and published a draft of ISO 10646 in 1990. This draft included multiple planes of characters and anticipated storage of millions of characters.

The TRON character set (first published in English in 1987 with work starting in Japan in 1984), was a sort of meta character set that could subsume existing encodings. TRON was a multibyte standard that included characters to switch between languages and a number of features to support Asian languages. The encoding allowed use of ISO 8859-1 (the most popular code page for English at the time), Braille, and multiple existing Japanese and Korean encodings.

Due to political and economic reasons, neither of these encodings won, with the former eventually aligning with Unicode and the latter being largely limited to hardware running TRON. Technically, though, they both anticipated using a multibyte encoding and the need for more than a 16-bit space.

6 IMPACT OF UNICODE ON STANDARD FUNCTIONALITY

In the Objective-C library reference, the documentation on strings states that a language only need to provide string length and character at index functions, since all the other functions can be built on these two primitives. Although it was once possible to have individual developers fill-in missing string functionality, with the adoption of Unicode, standard functions have become much more complex.

In terms of the primitives, string length is an ambiguous measurement due to the multi-byte encoding and semantic subtleties around the definition of a character. For instance, is the length a measurement of the buffer size, the number of individual code points, or the number of glyphs? Programmers need to understand the distinctions and how to apply them to their use cases. Languages such as Swift and Rust include distinct API calls to differentiate these states. String equality is similarly complex. Two byte sequences may differ yet yield equivalent Unicode sequences. Computing equality requires first normalizing the byte sequences; the documentation for this process runs over 30 pages. These complexities require a greater effort by language implementors to correctly handle the standard.

7 CONCLUSION

Programmers today can expect rich functionality and a high-degree of support for multilingual text from programming languages. Language designers face an increased burden to add initial support for these features due to the complex interoperability requirements, but continue to explore the performance and safety boundaries enabled by encapsulation of the string machinery.

ACKNOWLEDGMENT

We thank Bryan Short and Catherine Williams for providing time for this study.

References

[1] Accessed on 2021-08-18.

[2] James gosling on java, may 2001.

[3] String in std::string - rust.

[4] Ada reference manual. Technical report, International Organization for Standardization, 1995.

[5] Apple. Swift string, 2021. Accessed 2021-08-17.

[6] Joseph D. Becker. Unicode 88.

[7] Jim Brodie. Trigraphs: The search for alternatives. *The Journal of C Language Translation*, 3(4):310–318, 1992.

[8] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, 2000.

[9] Brad J Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[10] Emphrase. Source han sans version difference. Licensed https://creativecommons.org/licenses/by-sa/4.0/legalcode.

[11] R. A. Freiburghouse. The multics pl/1 compiler. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall), page 187–199, New York, NY, USA, 1969. Association for Computing Machinery.

[12] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., USA, 1983.

[13] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[14] Ralph Griswold. *The macro implementation of SNOBOL4; a case study of machine-independent software development.*

[15] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language.* Addison-Wesley Professional, 3rd edition, 2008.

[16] Michael Ilseman. Utf-8 string.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., USA, 1978.

[18] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, USA, 2018.

[19] Stefan Lucks, Norina Marie Grosch, and Joshua König. Taming the length field in binary data: Calc-regular languages. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 66–79, 2017.

[20] Conference on Data Systems Languages. *CODASYL COBOL: Journal of Development 1968.* National Bureau of Standards, United States, 1969-07.

[21] Rob Pike. Utf-8 history.

[22] Rob Pike. Strings, bytes, runes and characters in go, Oct 2013. Accessed 2021-08-05.

[23] P.J. Plauger. Character sets and c. *The Journal of C Language Translation*, 3(3):169–176, 1991.

[24] Dennis M. Ritchie. The development of the c language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 201–208, New York, NY, USA, 1993. Association for Computing Machinery.

[25] Keith Allan Shillington and Gillian M. Ackland. *UCSD PASCAL 1.5 Manual*, 1978.

[26] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., USA, 3rd edition, 1997.

[27] Guido van Rossum. Early language design and development.

[28] Guido van Rossum. Python release 1.0.1, 1994.

[29] Larry Wall. Perl kit, version 1.0, 1987.

[30] Laura Wideburg. Early years of unicode.