Matchertext: Towards Verbatim Interlanguage Embedding

preliminary draft - not yet for redistribution

Working project repository:
https://github.com/dedis/matchertext

Bryan Ford
EPFL

October 7, 2022

Contents

1 Introduction **2**

2 Background: needs and pitfalls of interlanguage embedding **4**
2.1 Special-purpose languages 4
2.2 The complexity of multi-level escaping . 5
2.3 A proliferation of quoting conventions . . 5
2.4 When security goes wrong 5

3 Matchertext design and rationale **6**
3.1 Abstract definition of matchertext 6
3.1.1 Escapeless embedding in matchertext 7
3.2 Concrete matchertext in practice 7
3.2.1 Standardizing on matcher pairs . . 7
3.3 Matchertext configuration variations . . . 8
3.3.1 Tightening variations 9
3.3.2 Loosening variations 9

4 Host language considerations **9**
4.1 General hosting considerations 9
4.2 C-like host languages 10
4.2.1 Some syntactic alternatives 10
4.3 SGML-style markup host languages . . . 11
4.3.1 Strings embedded as element content 12
4.3.2 Strings embedded as attribute values 12
4.3.3 Strings embedded in CDATA sections 13
4.4 Uniform Resource Identifiers 14
4.4.1 Precedents for URI syntax extensions 14
4.4.2 How liberal to liberalize? 15
4.5 Regular Expression Syntax 16

5 Embedded syntax considerations **16**
5.1 String literals in C-like languages 17
5.2 Comments and derived documentation . . 17
5.3 SGML-derived languages 18
5.4 Uniform resource identifiers 18
5.4.1 The near-matchertext-compliance of URIs 18
5.4.2 The URI end-finding problem . . 18
5.4.3 Matchertext resource identifiers (MRIs) 19
5.5 Regular expressions 19
5.5.1 Character classes 19

6 Implementations of matchertext **20**

7 Evaluation **20**

8 Related Work **21**

9 Conclusion **21**

Abstract

Embedding text in one language within text of another is commonplace for numerous purposes, but usually requires tedious and error-prone "escaping" transformations on the embedded string. We propose a simple cross-language syntactic discipline, *matchertext*, which enables the safe embedding a string in any compliant language into a string in any other language via simple "copy-and-paste" – in particular with no escaping, obfuscation, or expansion of embedded strings. We apply this syntactic discipline to several common and frequently-embedded language syntaxes such as URIs, HTML, and JavaScript, exploring the benefits, costs, and compatibility issues in adopting the proposed matchertext discipline.

1 Introduction

The need to embed valid strings in one language into valid strings in another is commonplace throughout programming practice. Just a few examples include embedding regular expressions, URIs, SQL queries, or HTML markup within string constants in general-purpose programming languages; embedding user-entered web form data or JavaScript variables into SQL queries; JavaScript code embedded in HTML via the `<script>` element; and URIs embedded as query strings within other URIs, such as in a query to a service like the Wayback Machine.

An equally-ubiquitous issue arising from this practice is the need to transform the embedded string – generally by *escaping* certain characters sensitive to the "host" language and syntactic context – so that host language processors will not misinterpret embedded text as host-language text. For example, the regular expression `"[^"]*"` matches double-quoted strings, but when embedded in a C-like language must be written like `re.match("\"[^\"]*\"")`, escaping all the embedded instances of double-quote characters, so that the embedded quotes will not prematurely end the string literal. Accidentally forgetting necessary escaping is naturally a common source of syntax errors in manual embedding practice. *Automated* embedding is also common practice, however, such as accepting an arbitrary user-entered string on a web form and embedding it into HTML via a scripting language like PHP. With automated embedding, forgetting to escape embedded strings properly has become an endless source of critical security bugs such as SQL injection [5]. or cross-site scripting attacks [10].

In some future evolution of today's standard programming languages and practices, could we achieve the ability to embed any valid string in essentially any language into any other – *across languages* – without ever having to escape, or otherwise transform or obfuscate, the string to be embedded? Could we make embedding always a simple matter of verbatim "copy-and-paste" when done manually, or a simple matter of concatenation or filling a "hole" in a tempate when done automatically? We propose that the answer can and should be *yes* – though with important challenges, costs, and caveats of course.

We observe that verbatim interlanguage embedding would be achievable if: (1) we could standardize across languages a set of open/close character pairs we will call *matchers*, such as the parentheses `()`, square brackets `[]`, and curly braces `{}`; and (2) we could impose the universal "syntactic discipline" that *matchers must properly match* in nested pairs throughout any valid string – without exception – in any compliant language. If *plain text* is an unstructured linear sequence of characters in a character set like ASCII or Unicode, then we define *matchertext* to be plain text conforming to the additional syntactic discipline that ASCII matchers must match. For example, the strings '`(a{b}c)`' and '`a({'}["])d`' are valid matchertext, while the strings '`(`', '`{a]`', '`[(])`', and '`}{`' are plain text but are not valid matchertext.

Consider *matchertext resource identifiers* (MRIs), a matchertext adapation of uniform resource identifiers (URIs) [1] or internationalized resource identifiers (IRIs) [7]. A URI like `http://my.site/path/` may always be transformed to or from equivalent MRI syntax like `http[//my.site/path]`. An MRI is embeddable verbatim, with no transformation, into another MRI or into another matchertext-aware language. Figure 1 shows two example search queries containing embedded resource identifiers, contrasting "copy-and-paste" embeddable MRI syntax with traditional URI syntax where the sensitive colon (`:`) and slash (`/`) characters must be escaped as `%3A` and `%2F%`, respectively.

Adopting the matchertext discipline does not eliminate the need for character escape sequences: in fact it can slightly increase the "escaping obligations" within a language, as discussed below. But matchertext enables lan-

```
URI syntax:   http://search.engine/linksto?site=http%3A%2F%2Fmy.site%2F&results=50
MRI syntax:   http[//search.engine/linksto?site=http[//my.site/]&results=50]

URI syntax:   http://historical.archive/get?site=http%3A%2F%2Fmy.site%2F&year=1998
MRI syntax:   http[//historical.archive/get?site=http[//my.site/]&year=1998]
```

Figure 1: Example queries containing embedded resource identifiers in URI and MRI syntax for comparison.

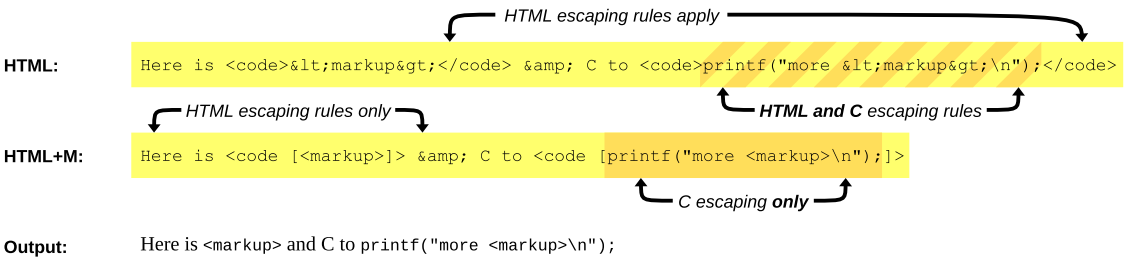

Figure 2: Illustration of how different languages' escaping rules combine to increase syntactic complexity in traditional embedding practice, while in matchertext only one language's escaping rules are ever active at a given text position.

guages to preserve the "territorial integrity" of their escaping and other syntactic rules, ensuring that developers need to think about *only one language's rules at a time* at any given text position, even in a string composed from multiple languages. Thus, matchertext arguably reduces the cognitive load of writing (or reading) cross-language embedded code. Figure 2 illustrates this difference with a simple example with C code embedded in HTML including the use of escape codes in both languages.

Existing languages could be adapted incrementally to support and leverage matchertext. C-like languages, for example, might adopt a new escape sequence like `\[`m`]` to embed an arbitrary matchertext m into a quoted string or character literal. All characters including quotes and newlines are allowed within m, provided only that matchers match. Thus, the earlier string-matching regular expression `"[^"]*"` could be embedded into a string literal like `re.match("\["[^"]*"]")`. With this extension, we use an escape sequence to delimit the entire embedded matchertext, but we no longer need to add escape sequences *within* the embedded regular expression.

Many languages already support *raw string literals* in which escape sequences are disabled: *e.g.*, backtick strings like `` `\` `` in Go. A particular terminating character or sequence must still be forbidden throughout the embedded text, however – in this case the backtick (`` ` ``). Verbatim copy-and-paste embedding remains unsafe without carefully checking the embedding text for the forbidden host-language delimeters, and changing the delimeters or awkwardly rewriting the embedded text (*e.g.*, by concatenating multiple strings in different quoting styles). Matchertext, in contrast, forbids *no* characters or sequences in embedded strings provided only that matchers match.

It is already feasible to write code in existing languages that is also valid matchertext, with a bit of care. Most languages use the matchers in structurally paired forms anyway, as in expressions like `a*(b+c)`, lists like `[a,b,c]`, or maps like `{a:1, b:"hi"}`. The challenge is mainly in handling the exception cases where unmatched matchers may commonly appear.

One traditional habit we must awkwardly unlearn in matchertext, unfortunately, is using unmatched matchers in quoted strings to parse or print structured text. Clauses like `printf("{")` or `case "]"` are not valid matchertext. We must therefore escape such unmatched matchers in literals, as in `printf("\x7B")` or

		Escapes for C-like languages						Escapes for SGML-derived languages			
		octal escapes		hex escapes		**matchers (new)**		entity names		**matchers (new)**	
		open	close	open	close	open	close	open	close	open	close
Parentheses	`()`	`\050`	`\051`	`\x28`	`\x29`	`\o()`	`\c()`	`&lpar;`	`&rpar;`	`&o();`	`&c();`
Square brackets	`[]`	`\133`	`\135`	`\x5B`	`\x5D`	`\o[]`	`\c[]`	`&lbrack;`	`&rbrack;`	`&o[];`	`&c[];`
Curly braces	`{}`	`\173`	`\175`	`\x7B`	`\x7D`	`\o{}`	`\c{}`	`&lbrace;`	`&rbrace;`	`&o{};`	`&c{};`

Table 1: Potential alternatives in C- and SGML-derived languages to escape unmatched matchers in matchertext.

`case "\x5D"` for example. Backward-compatible language extensions might ease this pain, however, with new escape sequences that include *both* matchers of a pair but select only the opener or closer. The proposed new escape `\o()` represents a literal open parenthesis, for example, while `\c[]` represents a close bracket. Table 1 summarizes a few existing and proposed alternatives for escaping unmatched matchers in both C-like languages and SGML-derived languages like HTML.

In the rest of this paper, we develop more deeply the design and rationale for the matchertext discipline, then explore how a number of common languages of varying types might be incrementally adapted to support and leverage the matchertext discipline effectively.

This work is in an early exploration and experimentation phase, so the evaluation is currently a placeholder, serving as a preliminary map for the ways in which we *would like to* evaluate matchertext and its use in practical languages. Some key questions we would like to answer include: how common (and how painful) is the need for escaping in the most common cross-language embedding scenarios? How extensively would large existing repositories of code or data need to be modified in order to convert them to matchertext? How does matchertext affect the usability to users or developers of common constructs in common embedding use-cases, such as synthesizing or editing HTTP or SQL queries? How does matchertext affect the frequency of syntax-related bugs – especially those potentially leading to security vulnerabilities – in code from typical developers?

An open research project

This draft represents a "work-in-progress" snapshot of an experimental open research project. Anyone with adequate interest, skills, and motivation is welcome to contribute to this research project, and potentially become a co-author upon making a substantial contribution. (Smaller contributions will receive acknowledgments in the final paper.) To propose a contribution, please use Pull Requests (PRs) on the project's GitHub repository. We do not have time and cannot promise to answer all E-mails, or provide detailed guidance, before an interested potential contributor has proactively created and submitted some significant and well-considered contribution.

2 Background: needs and pitfalls of interlanguage embedding

The practice of embedding strings from one language into another is ubiquitous – as is the pain of having to "escape" embedded strings to protect them from misinterpretation by the host language processor. This section briefly explores a few of these common existing practices and the syntactic composition problems they create.

2.1 Special-purpose languages

Many special-purpose language syntaxes exist almost solely for embedded use in other syntactic contexts. A few particularly common examples of such "little languages" include regular expressions (REs), uniform resource identifiers (URIs), JavaScript Object Notation (JSON), and Structured Query Language (SQL).

Perhaps the most classic "little language" is regular expression (RE) syntax, commonly used for pattern-based searches and replacements in freeform text. RE syntax traditionally uses many punctuation characters for special purposes, but must also allow arbitrary embedded text to be matched literally. RE syntax therefore makes heavy use of "backslash-escaping" in literal text: *e.g.*, the RE `.*` matches any number of arbitrary characters other than newlines, while one must write `\.\*` to match the literal

string '`.*`'. Because REs themselves are often embedded in another language – frequently a C-inspired language that *also* uses backslash-escaping in string literals – we must further backslash-escape the backslashes in RE syntax. A C string literal containing the latter RE above is written `"\\.\\*"`, for example. To match the double-backslash `\\` that begins a UNC name (as in `\\host\path\file`), the backslashes must be doubled to become `\\\\` as an RE, then doubled again to become `"\\\\\\\\"` as a C sring literal containing that RE. This confusing multi-level explosion of backslash escapes has been aptly dubbed *leaning toothpick syndrome*.

2.2 The complexity of multi-level escaping

As fig. 2 illustrates, each additional level of embedding in traditional syntax adds a set of escaping requirements that the writer (or reader) of code must carefully consider and apply correctly – *in the correct order* – in order to "guide" the embedded text through the levels of host syntax that the text is embedded in. REs and C string literals each require a *different* set of punctuation characters to be backslash-escaped, further increasing the cognitive load when embedding. PCRE syntax helpfully promises that a backslash followed by any non-letter always "takes away any special meaning that character may have", so one may fall back on just *escaping all punctuation* instead of remembering which characters must actually be escaped. But this practice feels like a band-aid at best, and yields even more "leaning toothpicks."

The above examples also illustrate how each level of embedding can multiply the length of embedded strings by a factor of 2 or more, yielding in the worst case an exponential string-length explosion with the number of embedding levels. While more than two levels of embedding may not be that common, they do occur.

2.3 A proliferation of quoting conventions

Escaping issues such as those above have in part led many languages to support multiple different types of quotes with different escaping rules. Many languages allow string literals to be either single-quoted or double-quoted, so that quote characters of one type can be used literally within string literals of the other, as in `"'"` or `'"'`.

Some languages disable escape sequences in one form of string literal so that backslashes may appear literally without multiplying in number: *e.g.*, single-quoted Bourne shell strings (`'\'` is the same as `"\\"`) or backtick-quoted raw string literals in Go (`` `\` `` is like `"\\"`). But without escapes it becomes more difficult to include the forbidden terminating quote literally in the string: typically one must compose multiple strings, like `"it's "+'"quoted"'`. Some languages such as Python allow triple-quoted multiline strings to make it less likely that the terminating sequence is needed in the literal: *e.g.*, `'''...'''` or `"""..."""`. But such a sequence may still need to appear, of course – especially in written examples of exactly this syntax for example.

Some languages further mitigate this problem by offering an effectively-unlimited number of delimiter pairs. Extended string literals in Swift, for example, surround a quoted sequence with a balanced number of `#` signs: *e.g.*, `#"..."#`, `##"..."##`, etc. Lua similarly offers *long bracket* quotations like `[=[...]=]`, `[==[...]==]`, etc. This approach has the appeal that for any string to be delimited, there always *exists* some delimiter pair that can quote it unambiguously. But the delimiters must still be carefully matched to the quoted string, or vice versa. It is still not possible to embed *any* string of a broad class verbatim into *any* "hole" or template in a host language without thinking about, and potentially adapting, either the choice of delimiters or the embedded string. Further, the worst-case "cost" of embedding in terms of string expansion still increases with each level of nesting – in this case at least only linearly, rather than exponentially, in the number of levels. We would prefer, however, if embedding required *no* expansion with increased depth.

2.4 When security goes wrong

The syntactic complexity of correctly embedding strings into code has created several broad classes of security vulnerabilities, where untrusted (typically user-entered) strings intended to be embedded can maliciously "trick" an application into interpreting parts of the string in the host language context.

SQL injection attacks [5], for example, typically arise from the common practice of embedding user-entered strings into string literals within SQL query templates. If a server composes an SQL query with a clause

like `"WHERE name='"+userName+"'"`, and the untrusted `userName` can be maliciously crafted to contain an unescaped single quote, then the attacker can prematurely terminate the SQL string literal and add other SQL clauses, like `OR '1'='1'` to make the clause unconditionally true regardless of `userName`.

Cross-site scripting (XSS) attacks [10] similarly exploit errors in the ubiquitous practice of embedding content from an untrusted source – such as fields from an HTML form – into HTML or other markup without proper "sanitization" via escaping. Suppose, for example, that one user of a Web-based discussion forum can post a message containing an HTML `<script>` tag – which the discussion server then inserts into corresponding pages viewed by *other* users of the forum. The injected JavaScript code can then potentially steal authentication cookies or other private information from *all* users on the site who might unwittingly read the maliciously-crafted message.

These and other broad classes of syntactic confusion attacks have led to the security-critical practice of sanitizing all potentially-untrusted user input before embedding it into security-sensitive code of any kind – whether SQL queries, HTML markup, or other host languages. The forms of sanitization needed in a particular context unfortunately tend to be complex and intricately dependent on the syntax and semantics of the host language that the untrusted content is to be embedded in. Version updates in the host language or associated libraries, which the application might not always track immediately, can easily introduce new syntactic attack vectors that the application has not yet countered with appropriate sanitization logic.

We do not expect any syntactic discipline, including matchertext, to eliminate the need to sanitize untrusted inputs. If a future SQL query or Web form is designed to accept embedded matchertext from an untrusted source, for example, then it will likely still be security-critical to check that the untrusted content *is indeed valid matchertext*, and reject it if not. However, a passive *verification* like this can be much simpler, and hence less bug-prone, than a content-modifying *transformation*, to escape all characters or sequences that might be "sensitive" in the host language. Further, this security-critical check could also be more uniform across host languages – *i.e.*, checking only that the three ASCII matcher pairs are matched corrrectly throughout the untrusted content, rather than deeply verifying and/or transforming based on the complex syntax of a particular host language. Thus, while matchertext will not eliminate the need for sanitization, it might tighten and simplify the function of the most security-critical "checkpoint" – namely checking that embedding content from an untrusted source preserves the structural integrity of the host language it is embedded in.

3 Matchertext design and rationale

This section first defines matchertext as an abstract mathematical syntax in section 3.1, then in section 3.2 as a concrete syntactic discipline pragmatically inspired by predominant practices. These definitions represent the "core" of the matchertext concept, and are the *only* rules that different languages must "agree on" in order to achieve the main goal of escapeless interlanguage embedding.

3.1 Abstract definition of matchertext

We first define *abstract matchertext* in order to ensure that the basic concept is clear and precise.

We assume at the outset we are given some arbitrary alphabet Σ, along with some finite set Π of character pairs $\{(o_1, c_1), \ldots, (o_k, c_k)\}$, such that $\{o_i, c_i\} \subseteq \Sigma$ for all $1 \leq i \leq k$. We define the *openers* O as the set of characters o such that some pair $(o, c) \in \Pi$. Similarly, the *closers* C are the characters c such that some pair $(o, c) \in \Pi$. We assume and require that the sets of openers and closers do not overlap: *i.e.*, $O \cap C = \emptyset$. We define the *matchers* M as the set of all openers and closers (*i.e.*, $M = O \cup C$). We define the *nonmatchers* N as all characters that are not matchers (*i.e.*, $N = \Sigma \setminus M$) A *matchertext configuration* is a pair (Σ, Π) following the above rules.

We now define the language L of *matchertext strings* inductively (*i.e.*, generatively) as follows:

- Any string n consisting exclusively of nonmatcher characters in N, including the empty string, is a matchertext string in L.
- For any matchertext strings m_1, m_2, m_3 in L, and for any pair $(o, c) \in \Pi$, the concatenation $m_1||o||m_2||c||m_3$ is also a matchertext string in L.

Intuitively, this definition captures the basic rule that *matchers must match* in matchertext. Openers can be introduced into valid matchertext only when paired with a

matching closer, and vice versa, but nonmatchers may be interspersed throughout with no constraints.

We consider matchertext to constitute a purely syntactic rule with no associated semantic meaning. While we can formally define it as a syntactic language, in practice we will refer to it as a *syntactic discipline* rather than a language because it assigns no specific meaning or purpose to the strings in L, and no structure apart from the basic rule that matchers must match. The meanings and structural purposes of all characters – both matchers and nonmatchers – are deliberately left entirely open for any particular "matchertext-aware" language to define.

3.1.1 Escapeless embedding in matchertext

Suppose there is a set of languages $\mathcal{L} = \{L_1, \ldots, L_k\}$ whose members all agree on a common matchertext configuration (Σ, Π). Any language $L_h \in \mathcal{L}$ can *host* embedded strings in any other language $L_e \in \mathcal{L}$ without escaping or other transformations to the strings from L_e, provided L_h enforces the following simple *embedding rule*. Any embedded matchertext string $m \in L_e$ must be delimited (surrounded) by some pair of strings s_o, s_c defined by the host language L_h, such that the full embedding sequence is $s_o||m||s_c$ in L_h. Further, s_o must contain one or more open matchers in O that would be unmatched in s_o alone – that is, s_o alone *cannot* be valid matchertext – and s_c must contain one or more corresponding close matchers in C that would be unmatched in s_c alone.

This embedding rule permits escapeless embedding because no fixed characters or strings, including s_o and s_c, need be unconditionally forbidden in the embedded string $m \in L_e$, provided that m is valid matchertext. The host language processor need not know anything about the embedded language L_e other than that m is matchertext. A host language processor that parses left-to-right from s_o can ignore any embedded instances of s_o and s_c within m because the matchers they contain must match within m. The host language processor can unambiguously recognize the closer string s_c that terminates the embedding because it contains at least one close matcher that would be unmatched, and thus illegal, if it were part of m.

In order to guarantee that verbatim embedding always works reliably, a host language not only can but *must* refrain from applying either transformations (*e.g.*, escapes) or restrictions (*e.g.*, disallowing certain characters or sequences) within the embedded matchertext it is hosting. We can view embedded matchertext as analogous to a diplomatic embassy, whose host country is required by international law to respect and protect the "involability" of the embassy's "premises" and not enter or otherwise meddle in the embassy's internal affairs [23, Article 22]. Beyond the matchertext rule that ASCII matchers must match, an embedded language's syntactic affairs are exclusively its own, not to be meddled in by a host language. While exceptions to this rule may sometimes be justified as we discuss below in section 3.3, any exceptions inevitably reduce verbatim embedding compatibility.

3.2 Concrete matchertext in practice

The abstract definition of matchertext above and its basic structural rule apply in principle to any matchertext configuration (Σ, Π). In practice, however, we must standardize on particular choices of Σ and Π across a set of languages of interest in order to achieve escapeless embedding among them. We wish to identify a particular, concrete matchertext configuration that fits existing syntactic syntactic practices as well as possible, and facilitates escapeless embedding across minimally-adapted variants of today's popular machine-readable languages.

We therefore propose a *standard matchertext configuration* whose alphabet Σ is the Unicode/UCS character set [25], and whose matcher pairs Π consist of the ASCII parentheses `()`, square brackets `[]`, and curly braces `{}`.

The choice of UCS as the character set Σ is justified by the fact that machine-readable languages have largely converged on this standard, so it is in effect already decided. In fact, the programming language community has also largely converged on UTF-8 as the standard way to encode UCS plain text into flat byte-stream source files – although encoding is not a primary concern for matchertext since it operates below the character set abstraction.

3.2.1 Standardizing on matcher pairs

A particular choice of the matcher pairs Π is less obvious, however, and hence demands more careful justification. We start by "deferring to authority": namely the authority embodied in the UCS character set we already chose. The parentheses, open brackets, and curly braces are the only characters in the ASCII – or "Basic Latin" – code

block that are standardized as members of the Open Punctuation (Ps) and Close Punctuation (Pe) character classes. Exactly as their official names indicate, these character classes denote characters whose standard purpose is to serve as open and close punctuation in matched pairs.

Why not the "angle brackets" <>? Many programming language also use the ASCII characters < and > in matching pairs, such as for generic types in C++ and Java, or markup in SGML-derived languages such as HTML and XML. These characters are not standardized as open/close punctuation, however, but as mathematical less-than and greater-than inequality symbols. Further, they are used for this purpose in mathematical expressions, in *unmatched* fashion, much more pervasively than their occasional use as matchers. Requiring these characters to be matched in matchertext would not only conflict with their primary standardized purposes (*i.e.*, would "defy the authority" of UCS), but would make it extremely cumbersome to express standard mathematical inequalities (*e.g.*, `a < b` in `if` expressions) in almost all programming languages. Omitting <> from the matched pairs Π of the standard matchertext configuration does not conflict with or prevent their paired use in specific languages – as we will see when we focus on SGML-derived languages later in section 4.3 and section 5.3. Omitting them from Π means only that we do not impose a "universal" rule that they *must* be used *strictly* as matchers, without exception, throughout all valid matchertext.

Why only the ASCII open/close punctuation? The full UCS standard of course includes much more open and close punctuation. UCS also includes Initial Punctuation (Pi) and Final Punctuation (Pf) character classes specifically for quotation marks intended for use in pairs (*e.g.*, «quote» or "quote"). None of these extended UCS characters are commonly used in machine-readable language syntax, however – no doubt in part merely by tradition, but also for the pragmatic reason that only the ASCII punctuation symbols are directly typeable on most keyboard layouts. Moreover, the open/close and initial/final punctuation in the extended UCS blocks do not occur strictly in pairs. For example, there are three different left double-quote characters (codes 201C, 201E, and 201F) that potentially match with the right double-quote character (code 201D), depending on linguistic culture and typographical style. Thus, deciding *which* character pairs should or should not match would become a much more complex question. While specific languages are free to use any UCS punctuation for their own language-specific structural or stylistic purposes, it seems simplest and safest to restrict the matchertext set Π of *strictly-matching* pairs to the ASCII matchers alone.

Why all three ASCII matcher pairs? We could of course be even more selective in choosing the set of matcher pairs Π. We could take only one matcher pair, for example: either parentheses *or* square brackets *or* curly braces. However, all of these matcher pairs are used quite pervasively, in a variety of different structural roles in different languages, and it is not readily apparent what principle would justify choosing one of these matcher pairs over the others to play a distinguished, globally-enforced matching role in matchertext. Moreover, interspersing multiple distinct matcher pairs in structured text in practice provides useful redundancy that helps detect errors more quickly and localize them more precisely. For example, it is much clearer where the missing close bracket is in the string `[{}([){}]` than in the similar but more homogeneous string `[[][[]]]`. Finally, any host language must use *some* matcher pair to delimit embedded matchertext strings, as discussed above in section 3.1.1. Including all three ASCII matcher pairs in Π thus gives languages maximum syntactic freedom in defining the syntax of matchertext embeddings (*i.e.*, a choice among three matcher pairs rather than a single prescribed pair).

3.3 Matchertext configuration variations

Even if adequately well-justified, we cannot expect the standard matchertext configuration as defined above to be a perfect or painless fit for all situations in which string embedding is useful. Including any matcher pair in Π has the cost of requiring those matchers to be escaped in string literals and comments, for example, as we detail later in section 5. There may be legitimate or even unavoidable reasons to use other matchertext configurations in some cases, keeping in mind that doing so reduces interlanguage embedding compatibility.

In general, deviations from the standard matchertext

configuration can be either *tightening* (more restrictive), *loosening* (less restrictive), or a combination.

3.3.1 Tightening variations

A matchertext configuration (Σ', Π') is a *tightening* of the standard matchertext configuration (Σ, Π) defined earlier if it only removes characters from the alphabet ($\Sigma' \subseteq \Sigma$) and/or makes additional matcher pairs sensitive ($\Pi' \supseteq \Pi$). A string in the tightened matchertext configuration may be copied verbatim to an embedding context expecting the standard matchertext configuration, but not necessarily in the other direction.

As we detail later in section 4.4, uniform resource identifiers (URIs) [1] traditionally allow only graphical characters – and no spaces or control codes for example – in order to make them manually transcribable. To serve this transcribability purpose, significant spaces and control codes must not appear *anywhere* in a URI, even in an embedded matchertext substring. Thus, URIs may represent a justifiable use-case for an alternate matchertext configuration that removes the non-graphical characters from the alphabet. This would unfortunately mean that a string cannot, in general, be copied from a standard matchertext language into a matchertext URI without transformation (*i.e.*, escaping spaces and control codes).

3.3.2 Loosening variations

A matchertext configuration (Σ', Π') is a *loosening* of the standard matchertext configuration (Σ, Π) if it only adds characters to the alphabet ($\Sigma' \supseteq \Sigma$) and/or removes sensitive matcher pairs ($\Pi' \subseteq \Pi$). A string in the standard matchertext configuration may be copied verbatim to an embedding context supporting the loosened matchertext configuration, but not necessarily in the other direction.

A loosened matchertext configuration might be justified, for example, if it is deemed critical to embed strings in some language L_e that frequently makes unmatched uses of some ASCII matchers, and the pain of escaping or otherwise adapting that syntax is deemed too great.

Mathematical notation, for example, sometimes uses "mismatched" parentheses and square brackets to represent half-open/half-closed intervals. That is, $[0, 1)$ typically means any real number r greater than or equal to zero but strictly less than one ($0 \leq r < 1$). A machine-readable language making frequent use of this mathematical notation might be considered too painful to embed in a standard matchertext configuration, and therefore might "demand" a looser configuration in which perhaps only the curly braces are sensitive as matcher pairs.

This example seems fairly hypothetical, however, as extremely few machine-readable languages appear to support this mathematical half-open/half-closed interval notation anyway. Languages that do support some form of half-open/half-closed syntax often do so with other, more matchertext-friendly notation. Swift [20], for example, supports *half-open range* syntax like `1..<4` for the sequence of integers starting from and including 1, up to but not including 4. This syntax is perfectly compatible with the standard matchertext configuration because it uses the mathematical inequality operators, rather than unmatched matchers, to express the range's open upper endpoint.

4 Host language considerations

This section focuses on considerations for, and potential extensions to, languages that may wish to *host* matchertext strings in other languages and provide the convenience of "cut-and-paste" embedding. Section 5 will later discuss considerations for languages wishing to *be embedded* conveniently. Both sets of considerations are relevant to languages wishing to be maximally "matchertext-friendly" of course. We present host-language and embedded-language considerations separately, however, in order to emphasize their conceptual orthogonality: a language could readily adopt hosting extensions but not embedding extensions, or vice versa.

4.1 General hosting considerations

Suppose a host language L_h wishes to allow embedding a matchertext string m from an arbitrary language L_e, whose syntax is likely unknown to the host language. In contexts where matchertext strings $m \in L_e$ are allowed, L_h must impose *no* constraints on characters allowed in that context other than the matchertext discipline (matchers must match). Further, L_h must not transform the embedded strings m in any way while extracting it from the host-language text. Specifically, any escaping mecha-

nisms or other transformations that might normally apply to text in L_h must be disabled in the context of the embedded string. If any escaping mechanisms or other transformations are active within the embedded string, they must be those of L_e, not L_h. We will see examples of this principle applied in several specific contexts below.

Languages need not *be* matchertext-compliant in their own syntax, however, just in order to *host* embedded matchertext. Existing languages can preserve full compatibility with all their existing (non-embedded, non-matchertext) code – continuing to allow unmatched matchers in string literals for example – while incrementally adding extensions that make it easy to embed matchertext strings verbatim within the host language. This form of backward compatibility will likely be essential to the incremental adoption of matchertext.

We next examine languages with C-like string literal syntax in section 4.2, then address SGML-derived languages such as HTML and XML in section 4.3, and finally in section 4.4 we focus on uniform resource identifiers in their role as a meta-syntax frequently "hosting" embedded identifiers derived from other syntaxes.

Table 2 summarizes the syntax extensions for different language classes proposed in this section. We emphasize that these are merely proposals for discussion. Different language communities should and will make their own decisions, and need not agree across languages on specific extension syntax in order for matchertext to be useful.

4.2 C-like host languages

An enormous variety of today's popular programming languages are derived, either closely or loosely, from C [21]. Though differing widely in purpose, philosophy, and semantics, a vast number of these C-inspired languages share similar syntax for string literals. In particular, most C-derived languages use double and/or single quotes to delimit a string literal, and backslash escape codes to insert "special" characters within the literal: *e.g.*, `"hello!\n"`. Because quoted string literals are the primary existing syntactic mechanism for embedding (non-matchertext) strings traditionally, they represent a natural starting point for considering matchertext extensions.

Given the ubiquity of backslash-escaped string literals, we suggest that one reasonable extension for hosting matchertext in C-like languages is via a new escape sequence, such as `\[`m`]`, where m is arbitrary matchertext. The embedded matchertext m is uninterpreted by the host language processor except to verify that ASCII matchers match and to find the terminating close bracket. Thus, quote characters, backslashes, whitespace, newlines, or other control codes cease being "special" within the matchertext m – at least from the perspective of the host language. For example, the string literal `"\["'\]"` becomes equivalent to `"\"\'\\"`. These and other characters might of course be "special" with respect to whatever embedded language m might be written in.

4.2.1 Some syntactic alternatives

The above proposal is only one of many possible alternatives of course, which may be worth considering – especially in the context of specific programming languages. We now briefly discuss a few "obvious" alternatives that seem less preferable for various pragmatic reasons.

The tradition of using quotes to delimit string literals is unfortunate in terms of matchertext's "cut-and-paste embedding" goal. Both the ASCII double quote (`"`) and the ASCII "single quote" (`'`) – technically standardized as an apostrophe and not a quote – are "undirected" and do not come in matched pairs, so C-style quoted strings do not naturally nest. Unicode offers directed quote characters intended for use in matched pairs, but they are harder to type directly on most keyboards, and are traditionally used in human-readable languages rather than programming languages. Also, the question of *which* Unicode quote characters go together is heavily language- and culture-dependent: *e.g.*, "English", „German", «French», »Danish«, etc. Thus, there is no obvious language-neutral way to choose and define a particular set of Unicode directed-quote characters as matcher pairs. Without doing that, quote characters are not useful to host embedded matchertext, because the "matchers must match" rule would not be sufficient for the host language processor to find the end of a matchertext string reliably.

Using ASCII matchers alone as new "matchertext string literal" delimiters – like `(`m`)`, `[`m`]`, or `{`m`}` – would obviously conflict with many other long-established and doubtless higher-priority syntactic uses, such as expression grouping `a*(b+c)`, tuples `(a,b)`, lists `[a,b]`, sets `{a,b}`, and maps `{a=1,b=2}`.

Nested *combinations* of quotes and ASCII matchers –

Class	Description	Syntax	Example	See
C-like	String escape	`"...\[`*m*`]..."`	`"now \[quoting's "easy" in matchertext]"`	4.2
⋆ML	Element	`<`*tag attrs* `[`*m*`]>`	`<code [if (a<b) { printf("some <markup>\n"); }]>`	4.3.1
	Attribute	`<`*tag attr*`=[`*m*`]>`	`<button onclick=[show("it's done!")]>OK</button>`	4.3.2
	Section	`<![MDATA[`*m*`]]>`	`<![MDATA[some example <b>bold</b> markup]]>`	4.3.3
URI	Bracket quote	`[`*m*`]`	`http://trans.late/?page=[http://my.site/]&lang=en`	4.4
	Percent escape	`%[`*m*`]`	`http://social.net/user/%[joe@email.net]/index.html`	4.4

Table 2: Summary of proposed matchertext hosting extensions.

such as `["`*m*`"]` or `"[`*m*`]"` or similar – might also be initially appealing. In most C-like languages, however, such combinations would similarly conflict with combinations of existing syntactic constructs that are not unlikely to appear in existing code: *e.g.*, a list whose sole element is a string literal, like `["x"]`, or a string literal containing brackets, like `"[x]"`. Using more deeply-nested matchers – *e.g.*, `[["`*m*`"]]` – only pushes these syntactic conflicts deeper (a singleton list of a singleton list of a string literal). Considering the other ASCII matchers (parentheses or curly braces) does not improve the situation much.

Embedding matchertext in a string literal via a new escape sequence also has the advantage that the *entire* literal need not be matchertext. Literals can mix matchertext with conventional literal text including host-language escape sequences: *e.g.*, `"\t\[let's indent]\n\t\[a "quote"]"`.

The choice of square brackets for the proposed matchertext escape sequence is somewhat arbitrary: we could instead use use parentheses or curly braces, or a longer sequence such as `\m[`*m*`]`. Any choice may conflict with existing syntax in *some* language: *e.g.*, `\(`*m*`)` conflicts with string interpolation in Swift, `\[`*m*`]` conflicts with octal character escapes in PHP, and `\{`*m*`}` conflicts with Unicode escapes in Ceylon. Fortunately, different languages need not agree on the precise syntax for hosting matchertext strings, and can choose whatever syntax best suits that particular language. To fulfill matchertext's main objective, different languages need to agree *only* on the basic rule that the embedded string itself is arbitrary except that ASCII matchers must match.

4.3 SGML-style markup host languages

While the venerable Standard General Markup Languages (SGML) [13,24] itself has waned in popularity, its derivatives HTML [27] and XML [28] are now ubuiquitous in Web content and programming. Wherever the differences between these markup languages is not important, we will refer to them all as ⋆ML languages.

In their basic role as markup languages used to produce rich, structured documents, ⋆ML languages frequently play "host" to embedded strings in countless other languages: typically, in the language(s) of software or APIs that a marked-up document is written about. Embedding code in other languages as verbatim text is a basic and frequently-used purpose of HTML's `<code>` and `<pre>` tags, for example. Beyond merely marking up verbatim text in other languages, however, HTML in particular has evolved to include special-purpose support for embedding several other languages within HTML: namely scripting languages such as JavaScript or Tcl, cascading style sheets (CSS) [31], MathML [29], and SVG [30].

The ⋆ML languages are surprisingly complex syntactically, especially given their simple-sounding purpose of "merely" describing structured markup of usually human-readable text. In particular, there are at least three different syntactic contexts in which strings in other languages are often embedded into ⋆ML languages – and in which three different sets of quoting and escaping rules apply. Embedded strings are often embedded (1) as the content of an element, (2) as an attribute within an element's start tag, or (3) as verbatim text within a CDATA section. We address each of these syntactic contexts in turn, in each case suggesting potential matchertext extensions that could help mitigate the various forms of "escap-

ing hell" that these embedding contexts can create.

4.3.1 Strings embedded as element content

One common form of embedding into ⋆ML is marked-up text serving as the content of an element: *e.g.*, example code between `<code>` and `</code>` tags or between `<pre>` and `</pre>` tags in HTML. Further, the `<script>` and `<style>` tags in HTML exist specifically to embed scripting language code and cascading style sheet (CSS) code, respectively, as their content.

The syntactic rules governing what can appear in text embedded as element content, however, depend intricately on the tag, the ⋆ML language in question, and even the language version. In most elements such as `<code>` and `<pre>`, any characters `<` and `&` appearing in the embedded string must be escaped (as `&lt;` and `&amp;`), to prevent the ⋆ML parser misinterpreting them as the start of a tag or a character reference, respectively. In XML, this rule applies to the content of all elements, including the content of `<script>` and `<style>` tags of XML-based XHTML. In HTML, however, the content of `<script>` and `<style>` tags is raw character data, uninterpreted by the HTML parser except to find the end tag. The content of such tags therefore *can* contain unescaped `<` and `&` characters – and *cannot* use HTML character entity references for escaping. In HTML4, this uninterpreted content is terminated by the first instance of a `</` character sequence, whether or not it is part of the corresponding end tag (`</script>` or `</style>`). HTML5 in contrast terminates the content with a sequence `</` followed by the appropriate end tag name. In all of these cases, figuring out what *must be*, what *can be*, and what *cannot be* escaped is subtle and potentially confusing.

As a potential extension enabling any of the ⋆ML languages to host embedded matchertext conveniently, we suggest the following new element syntax:

<name attributes `[`*matchertext content*`]>`

The *name* and *attributes* are the tag name and optional attributes as they normally appear in a start tag, and *matchertext content* is the element content as literal matchertext enclosed in square brackets, uninterpreted except to find the end by matching matchers. This syntax represents the entire element, with no end tag, so it is more concise than traditional start/end tag pairs. Since the content within brackets is uninterpreted except to match matchers, the content cannot contain further markup (child elements) or ⋆ML character entity references when using this syntax.

Figure 3(a) illustrates a few examples of embedding JavaScript into a `<code>` or `<script>` element, either in standard HTML or with the proposed matchertext content syntax (+M). The first and third examples embed trivial and non-problematic code. The second example shows the embedding of literal HTML markup within HTML. The fourth example illustrates the more troublesome corner case where embedded JavaScript wishes to output a `</script>` end tag within a string literal. Since HTML entity references are unavailable within a `<script>` element, the code must either use JavaScript escapes, or construct the `</script>` tag from two string literals, to prevent the embedded literal from prematurely ending the `<script>` element. In matchertext content syntax, neither example is problematic and both are more concise.

4.3.2 Strings embedded as attribute values

Besides element content, scripting language code is often embedded in the attribute values of ⋆ML start tags, most commonly to handle events in active user interface elements. Attribute values represent a different syntactic context in which different escaping rules apply. When attribute values are delimited with single or double quotes, the quote character that introduced the value must be escaped (as `&apos;` or `quot;`) if it is embedded in the attribute value. Character references may appear and are substituted in attribute values, like normal elements such as `<code>` in HTML but unlike `<script>` or `<style>` content. As the HTML specification notes, this means that script and style data cannot be simply cut-and-pasted between element content and attribute values without care for the changed escaping rules. HTML forgivingly allows `<` and "unambiguous" `&` characters to appear unescaped in attribute values, while XML requires them to be escaped (along with the active quote character).

One potential matchertext hosting extension would be simply to allow square brackets as a third "quoting style" for attribute values, where the text between the brackets is uninterpreted except to match matchers and find the end. With this extension as well as that above, the quoting

(a) Embedding strings in other languages as element content, in standard HTML or with matchertext hosting extensions (+M):

```
HTML  <code>printf("Hello world!");/code>
  +M  <code [printf("Hello world!");]>
HTML  <code>printf("Example &lt;b&gt;bold&lt;/b&gt; and &amp;bigstar; reference in HTML");]>
  +M  <code [printf("Example <b>bold</b> and &bigstar; reference in HTML");]>
HTML  <script>document.getElementById("demo").innerHTML = "Hello world!";</script>
  +M  <script [document.getElementById("demo").innerHTML = "Hello world!";]>
HTML  <script>document.getElementById("demo").innerHTML = "a <" + "/script> end tag";]>
  +M  <script [document.getElementById("demo").innerHTML = "a </script> end tag";]>
```

(b) Embedding strings in other languages within element attributes, in standard HTML or with matchertext extensions (+M):

```
HTML  <button onclick="okClicked()">OK</button>
  +M  <button onclick=[okClicked()]>OK</button>
HTML  <button onclick="emitCharacter('\'')">Emit Apostrophe</button>
  +M  <button onclick=[emitCharacter("'")]>Emit Apostrophe</button>
```

(b) Embedding strings in within CDATA (character data) sections, in standard XHTML or with matchertext extensions (+M):

```
XHTML  <code>example <![CDATA[<b>bold</b>]]> markup</code>
  +M   <code>example <![MDATA[<b>bold</b>]]> markup</code>
XHTML  <code>example <![CDATA[<![CDATA[character data]]]]><![CDATA[>]]> markup</code>
  +M   <code>example <![MDATA[<![CDATA[character data]]>]]> markup</code>
XHTML  <code>example <![CDATA[<![CDATA[<![CDATA[double embedded]]]]]]>
       <![CDATA[><![CDATA[>]]]]><![CDATA[>]]> markup</code>
  +M   <code>example <![MDATA[<![MDATA[<![MDATA[double embedded]]>]]>]]> markup</code>
```

Figure 3: Examples of embedded strings in standard ⋆ML languages, and with proposed matchertext extensions (+M).

and escaping rules for matchertext element content and matchertext attribute values would be identical, allowing code to be cut-and-pasted between these contexts freely.

Figure 3(b) illustrates script text embedded in attribute values, without and with matchertext hosting extensions. The second example illustrates how any time a string literal is needed in such embedded text, the embedding effectively "consumes" both quote characters in standard HTML or XHTML. Matchertext embedding, in contrast, preserves JavaScript's "syntactic freedom" of using one quote character to quote a verbatim instance of the other.

4.3.3 Strings embedded in CDATA sections

A third syntactic context in which strings are embedded in SGML and XML (but not HTML) is via CDATA sections of the form `<![CDATA[`*text*`]]>`, where *text* is mostly-uninterpreted character data. CDATA *sections* are distinct from CDATA-typed *entities* or *attributes* as declared in an SGML document type definition (DTD) [8]. CDATA sections offer the "greatest protection" from typical ⋆ML escaping requirements, in that *only* the section-terminator sequence `]]>` is disallowed within the embedded text. Because ⋆ML escape sequences are unavailable within CDATA sections, however, they also require the most-awkward syntactic contortions in the hopefully-rare event that a `]]>` sequence needs to appear in an embedded string. This "worst-case scenario" readily comes to pass whenever one is *writing about* CDATA sections and their issues in a ⋆ML markup language, for example.

A straightforward extension to host matchertext in a CDATA-like section would be simply to add a matchertext section form such as `<![MDATA[`*matchertext*`]]>`, where *matchertext* is uninterpreted matchertext. Figure 3(b) illustrates three examples of markup using CDATA sections versus corresponding MDATA sections. The first example is simple and non-problematic in either case. The second example illustrates how MDATA sections eliminate the problem of embedding a `]]>` sequence within such a verbatim section – provided that matchers still match, of course. The third example shows the more-extreme case of "double embedding" – where the complexity and visual obfuscation of CDATA sections explodes, while MDATA sections nest arbitrarily with no difficulty. This double-embedding scenario might seem

contrived, but it is exactly what is needed, for example, when attempting to write in ⋆ML markup a visual example (*e.g.*, in a `<code>` block) of the single-embedding problem and its typical "preferred" solution of replacing `]]>` sequences with `]]]]><![CDATA[>` sequences to "close and reopen" the outer CDATA section.

4.4 Uniform Resource Identifiers

Uniform resource identifier (URI) syntax [1] has become a ubiquitous notation for naming and locating not only web pages but innumerable other Internet resources. As a "small" special-purpose syntax, as opposed to a general-purpose programming language, it is arguably more often useful as an embedded rather than a host syntax, as we will focus on later in section 5.4. In practice, however, innumerable other identifier syntaxes get embedded into URIs regularly, either as scheme-specific text (*e.g.*, file names, phone numbers, cryptographic hashes), or even as query parameter values. This common and intentional use of URI syntax as a uniform "wrapper" for other identifier syntaxes makes URIs worth careful consideration as a potential host syntax for matchertext embedding.

We suggest two syntactic extensions to host matchertext within URIs, which are potentially complementary and could be adopted either together or individually.

First, extending the existing "percent-encoding" scheme for escaping special characters (*e.g.*, `%20` to represent an ASCII space), we suggest *matchertext escape* sequences of the form `%[`m`]`. Like the backslash-escape form `\[`m`]` suggested in section 4.2 for C-like languages, the matchertext m is uninterpreted by the URI processor other than to verify that matchers match and find the terminating close bracket. For example, `file:///%[a<b>c'd]` becomes a valid URI usable to access a local file named `a<b>c'd`, containing characters typically allowed in Unix-derived file systems but traditionally forbidden in URIs. Since percent-encoding by the host URI processor is disabled within the embedded matchertext m, `%[100%]` becomes valid and equivalent to `100%25`. In effect, this matchertext escape syntax offers a more concise, less obfuscated way to express arbitrary portions of URIs in which several characters would otherwise have to be individually percent-encoded. Figure 4 shows a few examples of URIs using conventional and matchertext percent-escapes for comparison.

Another potential syntactic extension is to allow square-bracketed sequences `[`m`]` to appear verbatim within the URI body, where m is otherwise-uninterpreted matchertext. This is *not* an escape sequence: the square brackets are not eliminated in URI processing, so `[@]` is equivalent to `%5B%40%5D`. This extension essentially serves as a matchertext-friendly quoting syntax that may be used in specific URI schemes, or within pathname components or query strings, to embed substrings in other identifier syntaxes (even other URIs) without obfuscation. We will explore the usefulness of this extension further when we consider *matchertext resource identifier* or MRI syntax later in section 5.4.3. This extension is backwards-compatible with existing (valid) URIs because current syntax permits brackets *only* in special-purpose IPv6 address syntax as part of the authority field.

4.4.1 Precedents for URI syntax extensions

Originally standardized as uniform resource locators or URLs [2], URIs traditionally allow only a small subset of ASCII characters to appear verbatim. Non-graphical characters, or those deemed "unsafe" for various reasons, must be escaped via percent-encoding. The set of allowed characters, and their purposes, has been "liberalized" multiple times historically, however.

IP version 6 introduced colon-separated hexadecimal addresses (*e.g.*, `1234::abcd` [17]), which conflicted with the URI's use of the colon to separate an IP address from a port number (*e.g.*, `http://1.2.3.4:80/`). The square brackets `[]` were therefore shifted from forbidden to "reserved" characters in URIs, for use *only* in embedding IPv6 addresses into the "authority" field of URIs, like `http://[1234::abcd]:80/` [16].

International Resource Identifiers or IRIs [7] further liberalized URI syntax, allow most of the graphical characters in the extended Unicode/UCS character sets to appear verbatim in URIs. IRIs preserved backwards compatibility in part by defining standard conversion processes back and forth between internationalized IRIs and legacy ASCII-only URIs. The set of ASCII characters allowed in IRIs remained tightly restricted, however.

Still later, IPv6 introduced scoped identifier syntax, allowing an interface number or name to be specified with an IPv6 address, *e.g.*, `1234::abcd%1` or `1234::abcd%if0` [6]. This new syntax again con-

```
URI   http://dev.site/myLibrary/doc/genericContainer%3CT%3E/api/
+M    http://dev.site/myLibrary/doc/%[genericContainer<T>]/api/
URI   http://search.engine/linksto?site=http%3A%2F%2Fmy.site%2F&results=50
+M    http://search.engine/linksto?site=%[http://my.site/]&results=50
URI   http://calculator.site/?expr=(1%2B2)*3%5E4%2F5
+M    http://calculator.site/?expr=%[(1+2)*3^4/5]
```

Figure 4: Examples of URIs without and with matchertext hosting extensions (+M).

flicted with URI syntax, leading to further syntactic hacks involving mandatory percent-escaping of the percent sign indicating a scoped identifier [3].

These points in in URI evolution illustrate a repeating precedent for liberalizing URI syntax to accept previously-forbidden characters and to make URIs more "friendly" and accommodating of embedded strings derived from other languages – whether machine-readable (*e.g.*, IPv6 addresses) or human-readable (international languages). Adopting matchertext hosting extensions such as those above, permitting URIs to host other syntaxes more cleanly without the traditional syntactic hacks and percent-encoding obfuscation, could be a useful step in allowing URIs to fulfill their ambition of being a uniform "meta-syntax" framework accommodating an unlimited variety of specific identifier syntaxes.

4.4.2 How liberal to liberalize?

The above considerations, however, raise the obvious question: *how far should liberalization of URI syntax go?* Beyond the square brackets, which *other* characters that were previously disallowed in URIs and IRIs eventually be permitted, and in what contexts?

In terms of our current focus on hosting embedded matchertext, the ideal would clearly be to allow *any* UCS characters in URIs – at least within a matchertext escape `%[`m`]` or a matchertext quote `[`m`]`. This approach would clearly provide the maximum latitude for embedding other syntaxes into URIs cleanly in the future. Further, only this "extreme liberalization" would guarantee that *any* matchertext, from any language conforming to the standard matchertext configuration (section 3.2), may be embedded verbatim into a URI without escaping.

As briefly discussed earlier in section 3.3, however, this arguably might be "going too far" in the case of URIs. We first consider the graphical ASCII characters that are currently disallowed in URIs, then the non-graphical characters such as spaces and control codes.

Graphical characters: URIs traditionally forbid "angle brackets" `<>` and double quotes `"` from use within URIs, because these characters are sometimes used to delimit URIs in surrounding freeform text: *e.g.*, `<http://my.site/>` or `"http://my.site/"`. In any "legacy" URI parsing context unaware of matchertext extensions, the appearance of these characters within a URI might indeed prematurely terminate the recognized URI, an issue we will return to later in section 5.4.2. In a context aware of the matchertext extensions, however, there is no syntactic ambiguity between angle-brackets or double-quotes used to surround a whole URI and any that may appear within embedded matchertext. The matchers delimiting the embedded matchertext unambiguously serve to differentiate "inside" from "outside": *e.g.*, as in `<http://my.site/%[>"<]>`.

The original URL standard in 1994 [2] additionally declared all the following characters to be "unsafe" in URIs "because gateways and other transport agents are known to sometimes modify such characters":

```
{ } | \ ^ ~ [ ] `
```

Nearly 30 years later, the "gateways and other transport agents" that text containing URIs tend to pass through have no doubt evolved drastically (or been replaced entirely). It is far from clear, therefore, that URIs today face the same modification perils as those of 1994. The text justifying the exclusion of these characters was dropped from the latest URI standard [1], though the characters themselves remain forbidden apart from the square brackets. IRIs introduced thousands of other new characters into the allowed set without dire consequences. With adequate care taken for backwards compatibility (as was done

with IRIs), it may be high time to consider allowing the rest of the ASCII printing characters above into URIs – *at least* in embedded matchertext hosted within URIs.

Non-graphical characters: URIs also traditionally disallow non-graphical characters such as spaces, as well as control characters such as newlines and tabs, for a different purpose: the goal of *transcribability*. It was difficult in 1994 to transcribe by hand a string containing spaces, newlines, tabs, or other control codes, and it is probably just as difficult to do so today. In an era of proliferating QR codes, the only question might be to what extent manual transcribability is still a crucial goal for URIs.

Nevertheless, manual transcription remains an important and not-uncommon use of URIs, and compromising the transcribability goal would arguably represent a much more fundamental shift in the principles underlying URIs than decisions about allowing or disallowing particular printing characters. This consideration therefore suggests that we cease the liberalization of URIs just short of allowing non-graphical characters.

Further, the transcribability goal is served only if the *entire* URI is readily transcribable, including any embedded matchertext substrings it may contain. Thus, as briefly mentioned earlier in section 3.3.1, the URI context may justify a *tightened* matchertext configuration whose alphabet Σ is reduced to exclude non-graphical characters. The cost is that arbitrary matchertext is not necessarily copyable verbatim from C-like languages into URIs, but this compatibility cost may be justified in this case.

4.5 Regular Expression Syntax

Despite being a "small" synax for the special purpose of matching patterns in strings, regular expressions (REs) and their use are complex enough in practice that multiple entire books have been written about them [9, 12, 14]. Two regular expression syntaxes were standardized by POSIX [19], while the Perl language and the Perl Compatible Regular Expressions (PCRE) library it inspired define advanced syntax that has become popular in numerous other languages and applications.

In filling their basic role of matching patterns in text, REs must inherently must embedd strings comprising the patterns to be matched – and those embedded strings can be in any syntax. Thus, being able to host embedded strings of any language with minimal obfuscation in principle facilitates an RE's basic pattern-matching role.

Since the most common RE syntax uses backslash escapes similar to those in C-like languages, the same matchertext escape extensions, such as `\[`m`]`, could work for REs. Because of the large number of punctuation characters that are sensitive in REs, however, some popular RE syntaxes such as PCRE guarantee the rule that if a backslash "is followed by a character that is not a number or a letter, it takes away any special meaning that character may have." This way, a user can just conservatively escape *all* literal punctuation appearing in an RE, instead of remembering which punctuation *must* be escaped. Preserving this rule may suggest a longer, letter-based matchertext escape sequence such as `\m[`m`]`.

Another alternative would be to use some non-backslash-escape syntax, such as `{{`m`}}`. This alternative syntax uses the same curly braces that REs already commonly use for repetition quantifiers like c{`1,3}`, but in a syntactically non-conflicting fashion since the inner braces cannot be mistaken for repetition quantifiers. With this alternative syntax, the pathological "leaning toothpick syndrome" example RE, matching the double-backslash `\\` in a UNC name (see section 2), becomes the more readable RE `{{\\}}`. Embedding this RE in a C-like string literal with matchertext extensions in turn becomes `"\[{{\\}}]"`, for a more manageable three leaning toothpicks total, in contrast with the traditionally-required eight (`"\\\\\\\\"`).

5 Embedded syntax considerations

Having focused above on host language considerations, we now switch focus to considerations for languages to *be embedded* as matchertext. The languages of interest for embedding overlaps heavily with those of interest as host languages; we separate these discussions mainly to emphasize the orthogonality of host- and embedded-language issues and cleanly separate them.

It is already readily feasible to write valid matchertext in most of the languages we will consider for embedding. This is because most popular machine-readable languages already largely conform to the "matchers must match" rule in their explicit uses of the matcher characters. Viola-

tions of the matchertext rule most commonly occur only in embedded "free-form" text such as string literals and comments. The language extensions we will propose are motivated almost exclusively by increasing convenience and visual clarity, and are by no means essential.

5.1 String literals in C-like languages

Almost certainly the most common context in which unmatched matchers appear in most today's existing source code is within string literals. This is especially true of code to print, or parse, machine-readable code in almost any syntax. Structured pretty-printing code frequently includes code sequences like this:

```
print("[")
output all elements of a list
print("]")
```

Similarly, parsing code often uses `if`, `switch`, or `case` conditionals to recognize and parse matcher-delimited syntactic structures, as in:

```
if peekNextChar() == '[':
  scanChar('[')
  scan all elements of a list
  scanChar(']')
```

Printing and scanning code like this generally violates the matchertext rule, and adapting such code most likely represents the biggest "pain point" in any venture to write readily-embeddable matchertext.

Almost all programming languages already offer a workable if slightly cumbersome solution: simply replace unmatched matchers in string literals with suitable numeric character escapes. Instead of `print("[")`, for example, write `print("\x5B")` (C, C++, JavaScript) or `print("\u005B")` (Java, JavaScript, Go). This always works; the main annoyance is that it requires the writer (and reader) of the code to remember or look up the codes for the matcher characters in an ASCII table.

The usual solution in C-like languages to handle "special" characters in string literals is simply to backslash-escape the special character, like `\[`. This traditional solution does not work for unmatched matchers in matchertext, however, because the matchertext rule is deliberately language-independent and oblivious to language-specific syntax such as that of string literals. So a backslash-escaped unmatched bracket `\[` remains just as much a matchertext violation as the bracket alone.

There is a solution that avoids the need for ASCII tables, however. Because literal matchers are a problem in matchertext only when unmatched, we can simply introduce escape sequences that incorporate *both* matchers as a properly-matched pair, while "selecting" only the opener or closer of the pair. In C-like languages, for example, we suggest the sequence `\o()` to escape an open parenthesis, `\c()` to escape a close parenthesis. Similarly, `\o[]` and `\c[]` represent open/close square brackets, and `\o{}` and `\c{}` represent open/close curly braces.

The choice of the letters `o` and `c` to escape the matchers is consistent with their standardized character classes: "Open Puntuation (Ps)" and "Close Punctuation (Pe)", respectively. We might consider `l` and `r` for "left" and "right", escept `\r` is a near-universal escape for carriage return (CR). A few languages already use `o` or `c` in escape sequences: *e.g.*, Raku uses `\o[`n`]` to denote the ASCII character with octal value n, and uses `\c[`n`]` to denote a Unicode character with name or decimal value n. Many of these existing uses are technically not in conflict syntactically, provided the existing use requires a non-empty string between the matchers – as Raku does in the above cases, for example. In any case, different languages need not agree on specific escapes sequences for unmatched matchers and are free to make their own stylistic choices.

5.2 Comments and derived documentation

Another context in which unmatched matchers may regularly appear in typical source code is within comments: *e.g.*, as part of human-readable text *describing* how the associated code handles particular characters. Conventional language processors usually just ignore unmatched matchers (along with everything else) in a comment. But the matchertext discipline operates below and oblivious to the syntax of a particular language, and hence does not know what a "comment" is – so the matchertext discipline must disallow unmatched matchers even in comments.

Since comments are generally intended for humans reading the source code, it is usually possible simply to rephrase the comment to avoid a literal use of unmatched matcher characters: *e.g.*, just name it ('open parenthesis') instead of writing it ('('). Another alternative, if a lan-

guage adopts the above extensions for string literals, is simply to use these matchertext-friendly escapes in comments as well (*e.g.*, `\o()`).

In some languages, comments often get used to produce API documentation, using tools like Javadoc or godoc. In such cases, it may be useful to interpret escape sequences such as those above while auto-generating documentation from source code, so that a documentation comment like '`// Parse a \o()`' becomes 'Parse a (' in the formatted output generated from the code.

5.3 SGML-derived languages

Considerations similar to those above for string literals apply when we wish to embed ⋆ML-language markup into other languages as matchertext. The most common reason unmatched matchers appear in markup is when needed in literal text being marked up: *e.g.*, human-readable text *about* the matcher characters or syntactic constructs built from them, or code examples that contain unmatched matchers.

As with C-style string literals, ⋆ML languages already offer a workaround: simply use character references, either named (like `&lpar;`) or numeric (`&#x0028;`). For the same reasons as above, we may like to have extensions offering more visually-obvious alternatives for writing matchertext: *e.g.*, `&o();` and `&c();` for open and close parentheses, respectively.

5.4 Uniform resource identifiers

Since uniform resource identifier (URI) syntax represents a special-purpose "little language" just for expressing identifiers, URIs are predominately embedded in other contexts – software source code, documentation markup, configuration files, etc. Especially since URIs are intended to be human-readable, it would thus seems useful if URIs could be maximally "friendly" for embedding.

5.4.1 The near-matchertext-compliance of URIs

Conventional URI syntax [1] already "nearly" complies with the "matchers must match" rule and is thus, usually, embeddable verbatim in a matchertext context. Curly braces are formally disallowed in URIs. Square brackets are allowed *only* to surround IPv6 addresses in the authority field, in properly-matched fashion. Thus, the only unmatched matchers that *can* exist in a strictly-valid URI are parentheses. Even these, when appearing in URIs, often still come in matched pairs anyway.[1]

In the rare cases when unmatched parentheses are "needed" in a URI, they may always be percent-escaped as `%28` or `%29`. For example, the string '`open(`' becomes '`open%28`' in a matchertext URI, '`close)`' becomes '`close%29`', and '`close)open(`' becomes '`close%29open%28`'. The string '`open(close)`' need not be rewritten at all in a matchertext URI, since the matchers it contains already happen to match.

We could always consider escaping extensions such as `%o()` and `%c()`, but it is far from clear that their likely-marginal need would justify the syntactic complexity in this case. Even if URI syntax is liberalized further to allow square brackets and/or curly braces in components, it is unclear how commonly unmatched matchers would be needed, since it is not particularly common to write parsing or scanning code within a URI for example.

5.4.2 The URI end-finding problem

Nevertheless, URI syntax does suffer from at least one significant usability flaw arising from its frequent use as an embedded syntax. URIs can and often do appear almost "anywhere" in freeform human-readable text – *e.g.*, typed or copied into E-mails, notes, documents, etc. Smart text editors often try to detect URIs on entry and automatically turn them into hyperlinks – but these heuristics can easily break because there is no unambiguous syntactic separation between the URI from surrounding (particularly following) text. Suppose for example that I type or copy this text into an E-mail:

```
My site is https://bford.info/index.html.
```

The trailing period (`.`) *could* be part of the URI, but in this case was probably intended to terminate my English sentence. I could try to "armor" the URI, like this:

```
See my site (https://bford.info/index.html).
```

But the close parenthesis, as well, *could* be part of the URI and be sucked into the link by a "greedy" URI auto-recognizer, resulting in a broken link. A careful reader of

[1] For example: `https://en.wikipedia.org/wiki/URI_(disambiguation)`

Appendix C of the URI specification [1] might find the recommendation to delimit URIs with angle brackets <> – but rather few people seem to be aware of this recommendation in practice, let alone are following it.

5.4.3 Matchertext resource identifiers (MRIs)

Given how commonly URIs are embedded in both freeform human-readable text as well as other machine-readable syntaxes of all kinds, we suggest that a more useful and ambitious potential evolution would make URI syntax *self-delimiting*. In particular, let us consider an alternative potential URI syntax in which we surround the URI's body – everything after the scheme name – with square brackets instead of separating it from the body with a colon. Thus, `http://my.site/` becomes `http[//my.site/]`. This alternate syntax uses only characters that are already used (and reserved) in current URI syntax, and it remains readily recognizable in freeform embedded contexts, but now the end can always be found unambiguously with no heuristic guessing.

Let's call this new syntax a *matchertext resource identifier* or MRI. Since MRI syntax is distinct and not readily confused with traditional URIs, it could enforce the rule that all URI body content within the brackets must be matchertext – *i.e.*, that unmatched matchers in the body must be percent-encoded – for verbatim embedding of other syntaxes (or other MRIs) in the body. Just as IRIs [7] liberalized URI syntax while preserving backward compatibility by defining automatic conversions in both directions, MRI syntax could similarly be converted automatically to or from traditional URI and IRI syntax.

Assume that MRI syntax includes the extensions discussed earlier in section 4.4 – in particular the rule that a square bracket sequence `[`m`]` nested within the URI body protects the embedded matchertext m from percent-encoding in the outer context. With this syntax, MRIs cleanly nest with no escaping needed, not even to introduce a matchertext embedding context. An embedded MRI appearing in a path or query string component of a host MRI never need be escaped, for example, as illustrated by the examples in fig. 1.

Moreover, MRI syntax could potentially be *simpler* than traditional URI syntax, because complex and rarely-used sub-syntaxes such as IPv4 and IPv6 addresses could be "broken out" of the main MRI syntax and handled instead as embedded MRIs in the host MRI's authority field. For example, the URI '`http://1.2.3.4:80/`' would become the 2-level MRI '`http[//ip4[1.2.3.4]:80/`', and the URI '`http://[1234::abcd]:80/`' would become the MRI '`http://ip6[1234::abcd]:80/`'. The MRI host field syntax thus knows only about domain names or nested MRIs, and not about IP address syntax.

5.5 Regular expressions

Typical regular expression (RE) syntax is C-like in terms of using backslashes to escape sensitive punctuation characters within text to be matched. Similar escape sequences like `\o` and `c` for unmatched matchers could therefore be introduced as discussed above in section 5.1.

One complication is that the popular PCRE syntax already uses `\o{`n`}` for character escapes with octal numeric value n. This octal-escape usage of `\o` technically does not conflict syntactically with `\o{}`, however, since the n in an octal escape cannot be the empty string.

PCRE syntax also offers `\c`x as a way to enter control characters, by flipping bit 6 of ASCII character x. This is a syntactic conflict with the proposed matchertext escapes, but perhaps a tolerable one. The sequence `\c(` would constitue a bizarre and unlikely way to express a simple literal letter '`h`', and `\c{` would be a strange synonym for a semicolon '`;`' – neither of which need escaping at all. The sequence `\c[` might be slightly more likely to see use to express the ASCII escape (ESC) control code (hex 1B) – but PCRE already provides the more-concise and obvious sequence `\e` to express this control code.

5.5.1 Character classes

Backslash escapes are normally disabled in bracketed RE character class notation like `[a-z0-9]`. The matchertext discipline does not present a problem when expressing a character class containing *both* matchers of a pair. For example, the character class `[()[]{}]` matches any matcher character, while `[^[]]` matches anything but a square bracket. Including just one unmatched matcher in a character class becomes less convenient, however. A slightly-cumbersome workaround is simply to shift unmatched matchers outside the character class: *e.g.*, `[a-z{]` might be rewritten as `([a-z]|\o{})`.

A more-appealing syntactic extension might be to introduce the rule that a less-chan character < in a character class, when immediately surrounded by a pair of matchers, "selects" only the open matcher for literal inclusion. The example above therefore becomes `[a-z{<}]`. A greater-than character > between a matcher pair similarly selects only the close matcher: `[^[>]]` matches anything but a close bracket. One might view the < or > character either as a matchertext-insensitive angle-bracket "standing in" for the desired sensitive matcher, or as an arrow "pointing" left or right to the desired matcher.

6 Implementations of matchertext

This section is mainly a placeholder at the present. Some preliminary work is underway implementing experimental matchertext extensions for several languages and embedding-oriented syntaxes. This section is intended to be expanded as we gain experience implementing and using matchertext extensions.

For those wishing to help with implementation and experimentation, the following are some of the key work items enabling us to start experimenting with matchertext in the context of any particular language of interest:

- A brief specification of proposed syntactic extensions for hosting matchertext and/or conveniently writing embedded matchertext, adapted to the specific language of interest, with clearly-defined rationale for the particular syntax choices.
- An implementation of those extensions for hosting and/or embedding, selectively enabled via configuration parameters for backward compatibility, in some mature processor for the language of interest (*e.g.*, a compiler, interpreter, or library implementation).
- A configurable extension to the language processor that optionally causes it to check and enforce the matchertext discipline in processed text: *i.e.*, to verify that matchers match in source files or strings.
- Purely for experimentation purposes, an extension of some processor for the language that can analyze "legacy" source files in the language to detect and categorize matchertext violations (*e.g.*, whether in string literals, comments, or elsewhere). We will use this to help estimate the likely "pain" of adopting the matchertext discipline in the language and how commonly this pain would affect typical code today.

7 Evaluation

This section is a placeholder at present, pending more implementation and evaluation experience to report.

Some key questions we wish to evaluate include:

- What are the most common kinds of embeddings that appear in large repositories of real source code, in what host and embedded language combinations, and for what purposes?
- How common and what kinds of needs are there for multiple levels of embedding in practice?
- For a set of popular (big or little) languages, how common are natural violations of matchertext discipline? Of what kinds are most common (e.g., in string constants, comments)? How painful would it be to fix these violations in typical code?
- How common and painful are needs to escape embedded strings when manually embedding into surrounding language strings? For example, how commonly do actual URIs embedded into actual program code need or use manual escaping?
- How common have security bugs been related to inadequate or incorrect escape armoring when embedding untrusted content automatically?
- What is the security-critical "attack surface" (*e.g.*, code size and complexity) of typical string sanitizing mechanisms for embedding of untrusted content?
- What are the syntactic "horror stories" of cross-language embedding, akin to leaning toothpick syndrome, but perhaps in other combinations of languages and/or resulting in other symptoms?

8 Related Work

The theory of syntactic structures, such as regular and context-free languages [4] or parsing expression grammars [11], has already been richly developed. Nothing about the matchertext discipline is particularly new or technically challenging from a formal language perspective. However, surprisingly little prior work has focused on the ubiquitous practice of synactically embedding strings of one language into those of another, or addressing the practical challenges this embedding creates.

Some recent work has focused on developing better tooling to support string-embedding practices as they currently stand: *e.g.*, parsing regular approximations of string-embedded languages [26], and support in integrated development environments [15] and static analysis tools [22]. This work does not attempt to explore syntax design practices that could make languages more cleanly embeddable in the first place, however.

Significantly more work has focused on *domain specific embedded langauges* or DSELs [18], particularly in the functional programming community. DSELs build upon the syntax and semantics of a general-purpose programming language such as Haskell, and thus benefit from – but also become dependent upon and specialized to – the syntax, semantics, and tooling of the host language. DSELs are thus unsuitable for embedded languages that wish to remain agnostic to, or usable across a variety of, host languages. Asking an embedded language to conform only to the matchertext discipline – that ASCII matchers must match – is a much more lightweight proposition than asking the embedded language to adopt, and become usable *only* with, the entirety of Haskell or another general-purpose programming language.

This related work is preliminary and no doubt incomplete; proposals for relevant additions are welcome.

9 Conclusion

Matchertext is a syntactic discipline that enables the verbatim embedding of strings across languages with no transformation or expansion, by enforcing only the single rule that the ASCII matchers – parentheses, square brackets, and curly braces – must always appear strictly in matched pairs. We develop the basic principles and rationale for matchertext, and explore potential backward-compatible extensions to existing classes of languages both to host embedded matchertext strings, and to write embeddable matchertext conveniently. Further experience is needed implementing and using matchertext in order to evaluate its benefits and costs empirically.

References

[1] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax, January 2005. RFC 3986.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL), December 1994. RFC 1738.

[3] B. Carpenter, S. Cheshire, and R. Hinden. Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers, February 2013. RFC 6874.

[4] N. Chomsky and M.P.Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 26:118–161, 1959.

[5] Justin Clarke. *SQL Injection Attacks and Defense*. Syngress, 2nd edition, 2012.

[6] S. Deering, B. Haberman, T. Jinmei, E. Nordmark, and B. Zill. IPv6 Scoped Address Architecture, March 2005. RFC 4007.

[7] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs) , January 2005. RFC 3987.

[8] Joe English. CDATA Confusion, September 1997.

[9] Michael Fitzgerald. *Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step*. O'Reilly Media, 1st edition, August 2012.

[10] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 1st edition, May 2007.

[11] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2004.

[12] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, third edition, August 2006.

[13] Michel Goossens and Janne Saarela. A Practical Introduction to SGML. *TUGboat*, 16(2), June 1995.

[14] Jan Goyvaerts and Steven Levithan. *Regular Expressions Cookbook: Detailed Solutions in Eight Programming Languages*. O'Reilly Media, second edition, September 2012.

[15] Semen Grigorev, Ekaterina Verbitskaia, Andrei Ivanov, Marina Polubelova, and Ekaterina Mavchun. String-Embedded Language Support in Integrated Development Environment. In *10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR)*, pages 1–11, October 2014.

[16] R. Hinden, B. Carpenter, and L. Masinter. Format for Literal IPv6 Addresses in URL's, December 1999. RFC 2732.

[17] R. Hinden and S. Deering. IP Version 6 Addressing Architecture, July 1998. RFC 2373.

[18] Paul Hudak. Modular domain specific languages and tools. In *5th International Conference on Software Reuse*, June 1998.

[19] IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) Base Specifications, January 2018. IEEE 1003.1-2017.

[20] Apple Inc. *The Swift Programming Language*. September 2022. Swift 5.7.

[21] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Pearson, 2nd edition, March 1988.

[22] Marat Khabibullin, Andrei Ivanov, and Semyon Grigorev. On Development of Static Analysis Tools for String-Embedded Languages. In *11th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR)*, pages 1–10, October 2015.

[23] United Nations. Vienna Convention on Diplomatic Relations. *Treaty Series*, 500, April 1961.

[24] International Standards Organization. Information processing — text and office systems — standard generalized markup language (SGML), 1986. ISO 8879:1986.

[25] International Standards Organization. Information technology — Universal coded character set (UCS), 2020. ISO/IEC 10646:2020.

[26] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. Relaxed parsing of regular approximations of string-embedded languages. In *Perspectives of System Informatics (PSI)*, August 2015.

[27] WHATWG. HTML Living Standard, September 2022.

[28] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008.

[29] World Wide Web Consortium. Mathematical Markup Language (MathML) Version 3.0 2nd Edition, April 2014.

[30] World Wide Web Consortium. Scalable Vector Graphics (SVG) 2, October 2018.

[31] World Wide Web Consortium. Cascading Style Sheets, December 2021. CSS Snapshot 2021.