Gramble: A Tabular Programming Language for Collaborative Linguistic Modeling

Patrick Littell[1], Darlene Stewart[1], Fineen Davis[2], Aidan Pine[1], Roland Kuhn[1]
[1] National Research Council Canada, [2] Congress of Aboriginal Peoples
1200 Montreal Rd. Ottawa ON, 867 St. Laurent Blvd. Ottawa, ON
{Patrick.Littell, Darlene.Stewart, Aidan.Pine, Roland.Kuhn}@nrc-cnrc.gc.ca, fineen.davis@gmail.com

Abstract

We introduce Gramble, a domain-specific programming language for linguistic parsing and generation, in the tradition of XFST, TWOLC, and Kleene. Gramble features an intuitive tabular syntax and supports live group programming, allowing community experts to participate more directly in system development without having to be programmers themselves. A cross-platform interpreter is available for Windows, MacOS, and UNIX, supports collaborative programming on the web via Google Sheets, and is released open-source under the MIT license.

Keywords: finite-state transducers, morphology, parsing

1. Introduction

Language technology for "very-low-resource" languages is usually talked about in the future tense, e.g., "techniques like these may someday allow for language technologies for all languages." However, there is a considerable amount already available: spellcheckers, orthography converters, search tools, conjugators, etc., usually based on a core of handwritten declarative code rather than machine learning (Arppe et al., 2016; Littell et al., 2018a; Kuhn et al., 2020). Many were never associated with publications and fly under the academic radar; they are analogous to what Hanselman (2012) calls the "dark matter" of programming.

Handwritten declarative programs still have advantages even in the "AI" era, particularly for educational applications like verb conjugators where there are known correct answers. We do not want students acquiring neural fabrications; a beginner student cannot necessarily catch the lie! Meanwhile, teachers feel strongly that they should have the ultimate say over system outputs, which cannot be guaranteed when decisions are made inside uninterpretable black-box language models.

However, we should also be clear-eyed about issues developers and their clients face when developing rule-based models. Especially for morphologically complex languages, the source code tends to balloon in complexity to the point that it becomes difficult for anyone except the original programmer to read and maintain it, making the system another sort of black box. What can we do to mitigate this?

1.1. Source Code Accessibility and Language Sovereignty

Our team largely serves community organizations and schools, in languages without ML-scale data resources, and most of our products still incorporate handwritten declarative components at some level. Some are in domain-specific languages (DSLs), others in Python or other general-purpose languages, others in ad-hoc "little languages" (Bentley, 1986). For example, the verb conjugators Kawennón:nis (Kazantseva et al., 2018) and Li Verb Kaa-Ooshitahk Di Michif (Davis et al., 2021) were backed by handwritten finite-state transducers (FSTs) in XFST/Foma (Beesley and Karttunen, 2003; Hulden, 2009). Even systems that utilize ML, like Littell et al. (2022), rely on human-written rules to bridge the gap for sub-tasks where training data is unavailable.

As our collaborations evolved beyond proofs-of-concept and student projects, source-code complexity became a growing concern for some community collaborators: "We appreciate the effort, but we can't make heads or tails of the code. If we need to change something, who is going to do it? What if the original programmer moves on?"

This is a real but under-discussed issue of language sovereignty: how dependent is the language community on a third party for system updates, changes, and maintenance?

Many of these are systems for education, and a product that cannot change along with the curriculum is effectively dead. This parallels an observation by Reiter (2021) that natural-language generation systems tend to fall out of real-world use after a few years, partly because users found them "hard to configure, tweak, or otherwise modify... such changes could only be made by the software developers."

It became clear that it was not just an issue of the choice of any particular programming language, but the nature of the collaboration. Subject-matter experts (SMEs) like teachers and linguists would send the programmers knowledge like root lists,

LREC-COLING 2024, pages 7913–7925
20-25 May, 2024. © 2024 ELRA Language Resource Association: CC BY-NC 4.0

paradigm tables, etc. (usually as an Excel spreadsheet or Google Sheet), and the programmers would turn those into code. But this transformation is irreversible, and effectively cuts the community out of further development. The "real" project is no longer the experts' knowledge but a set of dense source files in a world of command line interfaces, code editors, version control, and other systems that have a high barrier to entry.[1]

1.2. Three Components of a Solution

Over the course of these projects, we made three breakthroughs that, together, made it substantially easier for us to involve SMEs in the development of rule-based linguistic programs, and early adopters reported significant productivity increases.[2]

1. On the modeling side, we adopt a more flexible transduction model, generalizing Brzozowski derivatives to multi-tape regular expressions (§2).
2. We map the resulting expressions onto a tabular format that is easier to read and write, even for non-programmers (§3).
3. We embed an interpreter/IDE for the language into a Google Sheets add-on that enables live pair-programming between the programmers and subject-matter experts (§4).

The result is a language in the tradition of TWOLC (Koskenniemi, 1983, 1986), XFST (Beesley and Karttunen, 2003), and Kleene (Beesley, 2012). It is likewise intended for the development of applications like interactive verb conjugators (e.g. Kazantseva et al., 2018), spellcheckers (e.g. Arppe et al., 2016), morphological analyzers (e.g. Strunk, 2020), text completion systems (e.g. Lane et al., 2022), orthography and grapheme-to-phoneme converters (e.g. Mortensen et al., 2018), and other rule-based programs.

The interpreter is written in TypeScript so that it can run entirely client-side, and is released under the MIT license at github.com/nrc-cnrc/gramble.

[1]The obvious objection here is, "Why not teach the SMEs these systems? Everyone should learn computer science!" But the typical SME we encounter is a busy teacher or other language professional juggling other important projects. Even when they want to become more involved in the technical side, they simply cannot afford the time to re-skill in traditional computer science.

[2]Early adopters reported that they felt about 10x more productive compared to working in XFST. This probably would not bear out if we measured it with a stopwatch, but it is a testament to just how much they preferred working in the new environment.

VRoot =	text/root	eng	valence	
	pend	love	trans	
	on	see	trans	
	end	go	intrans	
TenseStem =	text	tense	embed	text
	na	PRES.CONT	VRoot	a
	a	PRES.INDEF	VRoot	a
	li	PAST	VRoot	a
PersonStem =	text	subj	embed	
	ni	1SG	TenseStem	
	u	2SG	TenseStem	
	a	3SG	TenseStem	
replace text:	from	to	context	
	u	w	#_a	
	a		#_a	
test:	text	subj	tense	
	ninapenda	1SG	PRES.CONT	
	wapenda	2SG	PRES.INDEF	
	uapenda	2SG	PRES.INDEF	

Figure 1: A toy grammar of Swahili verbs in Gramble, illustrating multiple tapes (text, root, eng, valence, tense, and subj), symbol embedding, phonological rewrite rules, and successful (green) and failed (red) test forms. Note that this is not just an input spreadsheet to be turned into code later; this is the actual source code and represents a complete project in itself.

1.3. Accessibility to Non-Programmers

The primary differentiator of this language, named Gramble (a portmanteau of *grammar+table*), is its focus on accessibility to non-programmers and features that better enable group coding (e.g., namespaces, built-in unit test capabilities, local isolation of syntax errors, and immediate interpretation without re-compilation). The goal is that beginner and non-programmers can meaningfully contribute without being excluded by common barriers to entry, and realistically be able to inherit and maintain the project if the original programmer moves on.

Gramble has already been used to make educational software in eight languages, ranging in scale from simple activities for elementary-school students to university-level reference tools covering more than a million forms. For example, the Gramble port of LVKODM (Davis et al., 2021) grew to cover more, and more complex, phenomena than the XFST original. The original had grown to roughly the maximum complexity a human linguist-programmer could practically manage, but the greater ease in reading and organizing code in Gramble allowed the programmer to get past this complexity wall and cover additional phenomena.

1.4. A Tabular Programming Language

The other differentiator of Gramble is that it is a tabular programming language akin to Gordon et al. (2014), where the spreadsheet describing the phenomenon and the code are the same document.

It is important to note that this is not "programming by spreadsheet"; this is not attempting to handle complex phenomena by an awkward soup of Excel macros. It is simply a language in which source documents are grids of cells, in the same way that the canonical form of a Python script is plain text or a Node-RED project is a flow diagram. The source files are not tied to the use of any particular editor; they are simply CSVs.

The tabular syntax is designed so that knowledge workers with a basic familiarity with spreadsheets and databases can understand and contribute. While their semantics is not exactly the same, a plain CSV file can often already be interpreted as Gramble code as-is.[3]

2. Modeling Multiple Fields

There is an irreducible complexity to a description of a complex language, whether you are describing it in English or XFST or Python. However, there is also complexity added by the programming language itself: complex syntax, awkward constructions, and workarounds for missing features.

2.1. Issues with Flags

XFST and similar languages allow the expression of complex linguistic phenomena by composing simpler two-tape relations. This has the benefit of conceptual simplicity and efficiency when compiled, but many linguistic phenomena need access to additional fields of information (e.g., that a certain affix or rule only applies when a particular morphological feature has a particular value). To handle this, it is often necessary to interleave the additional field into one of the tapes, either by ad-hoc markup (e.g., placing morphological labels into content that is otherwise phonological) or by the use of "flags".

Flag diacritics are built-in workarounds in XFST, in which the programmer can specify custom tokens that cause a transduction to fail if incompatible tokens (e.g. `@U.TENSE.PAST@` vs. `@U.TENSE.PRESENT@`) are encountered on the same tape. This allows the programmer to handle long distance co-occurrence restrictions or the selective application of affixes and rules.

Flags are typically necessary to model morphologically complex languages (e.g. Bosch and Pretorius, 2006), but their syntax is verbose and can begin to overwhelm other text content (§3.1), and they can be difficult to reason about because they often describe long-distance dependencies with potentially thousands of lines in between them. Also, since they are hidden characters embedded into otherwise-ordinary strings, they can make phonological rules more difficult to write by silently destroying adjacency.

2.2. Issues with Two Tapes

Another issue is that limiting the number of tapes limits the types of queries possible, and thus limits the possibilities for the model's flexible re-use. This may seem minor but it is another source of ballooning complexity as we try to adapt FST-based systems to additional languages and additional tasks.

It is typical that a morphological FST treats the lower language as a surface/orthographic form and the upper language as a morpheme breakdown or Leipzig-style gloss. But there are many possible kinds of information one might put in the latter. Do you use the underlying forms or UniMorph-style (Kirov et al., 2018) labels? Do you use the native root or an English translation? Are discontinuous morphemes labeled in one place or two? What about structural elements without clear meanings?

It is thus rather difficult in practice to take off-the-shelf FSTs for different languages, even related ones, and use them together without significant adaptation. In Schwartz et al. (2020), for example, many of the target languages already had mature FST parsers available, but the upper languages were too diverse in their conventions to be useful for cross-linguistic modeling.[4]

This can be true even when parsing a single target language, for example in Littell et al. (2018b) where different downstream tasks required different information from parses. Building an FST to generate one of these representations and then building systems to adapt that into the other forms is possible, but would require some information *in* the system to be recapitulated in the adaptor.

Similar problems arise when generating surface forms from glosses. The client must construct a sentence of the upper language, and thus must know what morpheme labels are possible and how exactly the FST expects them to be ordered/formatted. This is further complicated when client systems like paradigm generators need to make

[3] As an anecdote, we once set aside an afternoon to convert a linguist's informal paradigm table to a working Gramble program, and found that it was so close to Gramble code already that we had a working prototype in ten minutes.

[4] Or, in some cases, the FST source *does* have the same information as other languages' systems, but only in source comments (e.g., an English label as information for the programmer). The information is *there* but cannot be accessed by client programs.

"wildcard" queries like "Give me all forms where the root is X and the subject is Y, but the tense and object can be anything." This cannot be structured as a single query in an FST because wildcards are not part of the upper language; the client has to know all the tense/object possibilities, construct glosses in the expected format, and transduce each in turn.

For morphologically-complex languages, ordering and co-occurrence restrictions can themselves be quite complicated. In trying to adapt Kazantseva et al. (2018) into a language-neutral client that could work with FSTs for multiple languages, we found that we would be duplicating grammar knowledge in adaptor code that should, ideally, be expressed only in one place, in the grammar itself.

Given these experiences, we decided that the ability to handle multi-field to multi-field queries was a priority, not through workarounds like interleaving, but by assuming n-tape automata at the outset.

2.3. Multi-tape Automata

Multi-tape automata (Rabin and Scott, 1959; Elgot and Mezei, 1965) with $n > 2$ tapes have been used to address various phenomena in linguistic description where two-tape FSTs are inadequate or awkward. For example, Kay (1987), Habash and Rambow (2006), and Kiraz (2000) use them to express nonconcatenative morphology, Wiebe (1992) to express tone, and Hulden (2017) to recover the intermediate forms of phonological rewrites that otherwise would be lost in a two-tape transduction.

In this work, we use multi-tape automata to avoid the additional syntactic and semantic complexity of interleaving and flag diacritics, and to enable client interfaces to construct queries and receive outputs using any combination of fields. Rather than encapsulating additional fields into special characters on the input/output tapes, they are treated as first-class tapes in their own right.

This allows the programmer to associate an arbitrary number of fields with each entry, rather than being restricted to two, and to express constraints that likewise reference any number of fields. Also, since these additional fields are first-class strings like any other, the constraints can be expressed in terms of arbitrary regular expressions on those fields (i.e., constraints are not limited to the presence or absence of atomic flags).

Put another way, this solution generalizes string-to-string transduction to "dict-to-dict" transduction. In a typical morphological FST, we can transduce either from a gloss to a surface form (e.g., inputting `1SG-PR-pend-V` and receiving `ninapenda`), or vice-versa. In Gramble, that remains possible, but you can also look up these forms from any combination of other fields. For example, a verb conjugator could generate the text from an unordered set of query fields `{root:pend, subj:1SG, tense:PR}` without constructing them into a gloss, or one could query `{text: ninapenda}` and receive the other fields in return.

When a field is missing from a query it is treated as a wildcard. For example, a query consisting only of `{root:pend}` returns all entries with this root; the empty query `{}` returns all entries.

The API is thus essentially that of a NoSQL database, greatly simplifying lookup operations for clients like the paradigm generators discussed in §2.2. We achieve this with a multi-tape regular expression evaluator that can be queried using any field or combination of fields.

2.4. Brzozowski Derivatives

Brzozowski (1964) introduced an alternative DFSA construction algorithm based on derivatives of regular expressions.[5] The algorithm remained rather under-appreciated and under-cited in the parsing literature until its rehabilitation by Owens et al. (2009) and Might et al. (2011), who showed that despite its very poor worst-case runtime complexity when unoptimized ($O(2^{2^n}G^2)$ where G is the size of the grammar), an optimized version can approach $O(nG)$ in practice, and constructs more compact graphs than many better-known algorithms.

Brzozowski presented a principled way to add additional operators to regular expressions, and subsequent work has shown that it also straightforwardly extends to context-free languages (Might et al., 2011), and even to objects that are not strings, like trees (Attou et al., 2021). All that is necessary is to define two functions for each new operation[6]:

- A derivative function D_cL that returns the language consisting of strings in L that begin with c, with c removed from the beginning (Figure 2a). E.g., if L denotes a language equal to the set `{"abc", "bobo", "b"}`, D_bL denotes `{"obo", ""}`.
- A nullability function δL, which returns the trivial language ϵ (consisting only of the empty string) if L contains the empty string and $\emptyset$ otherwise (Figure 2b). E.g., if L denotes `{"abc", ""}`, δL denotes `{""}`.

One then specifies algorithms in terms of these two functions, allowing the expression of lookup and compilation algorithms in a polymorphic way, agnostic to the exact types of the objects on which they are operating.

[5] Antimirov (1995) introduced partial derivatives of regular expressions; internally the Gramble interpreter mostly uses partial derivatives for efficiency until operators are encountered that require the full derivative, such as negation. We present the Brzozowski formulation in Figure 2, however, for clarity.

[6] We follow Might's (2011) naming conventions here.

$$D_c\emptyset = \emptyset$$
$$D_c\epsilon = \emptyset$$
$$D_c a = \begin{cases} \epsilon, & \text{if } c = a \\ \emptyset, & \text{otherwise} \end{cases}$$
$$D_c(A{\cdot}B) = D_cA{\cdot}B + \delta A{\cdot}D_cB$$
$$D_c(A+B) = D_cA + D_cB$$
$$D_c(A*) = D_cA{\cdot}A*$$

(a) The derivative function D_cL returns the members of L that begin with c, when c has been removed.

$$\delta\emptyset = \emptyset$$
$$\delta\epsilon = \epsilon$$
$$\delta a = \emptyset$$
$$\delta(A{\cdot}B) = \delta A{\cdot}\delta B$$
$$\delta(A+B) = \delta A + \delta B$$
$$\delta(A*) = \epsilon$$

(b) The nullability function δL returns the trivial language ϵ when L contains the empty string and $\emptyset$ otherwise.

Figure 2: Brzozowski's derivative for Kleene regular expressions

A simple lookup algorithm allows membership tests even without compilation to a DFSA. One can look up whether "abc" is in L by evaluating $\delta D_c D_b D_a L$; if the result is ϵ the word is present, otherwise it is not. One can also construct a DFSA by generating the graph of all unique derivatives.[7] The lookup algorithm can be viewed as the lazy construction of a DFSA, only creating nodes and transitions necessary to evaluate the query at hand.[8]

2.5. Generalizing Derivatives to *n* Tapes

Brzozowski's algorithm has several advantages for our situation. Just as it is possible to generalize these formulae beyond strings to trees, it also proved possible to generalize them to arbitrary sets of ⟨key,value⟩ tuples. Our generalized formulae only ended up differing from the single-tape case in a few details.

The "lazy" construction of the graph is also an advantage. In a two-tape FST, we can compile once and support both query directions (upper↔lower) using the same graph. Trying to extend this flexibility to any number of tapes, with any of them being the input tapes, would result in an explosion in the number of nodes and transitions needed. Instead, we can construct only the nodes and transitions necessary to evaluate the provided query.

We generalize the one-tape formulation in Figure 2 to n tapes in Figure 3. A language L is no longer a set of strings, but a set of sets of ⟨key,value⟩ tuples; for brevity we will denote the tuples as $K{:}v$. When a key is undefined in an entry, we stipulate that it contains the empty string (that is, `{root:"pend"}` and `{root:"pend", subj:""}` are identical). The trivial language ϵ is now the set of empty sets – i.e., the language with only one entry that has an empty string for all keys.

The first, minor change to the equations is that string literals, D, and δ are now relative to a particular tape T.

The crucial change lies in the semantics of δ. Brzozowski's δ is effectively a boolean return: we could as easily return T and F as ϵ and $\emptyset$.[9] This formulation would not work, however, when there are additional tapes beyond T; it would "drop" material on those tapes as soon as the end-of-string on tape T was reached. Instead, material on unrelated tapes must be preserved; that is to say, $\delta_T X{:}a$ must not just return ϵ or $\emptyset$ when queried on an irrelevant tape, but return $X{:}a$ intact.

Consider the simple multi-tape regular expression $A{:}a{\cdot}B{:}b$, and the question of whether the entry `{A:a,B:b}` can be generated. If we adopt a naive solution identical to the single-tape formulation, where $\delta_T X{:}a = \epsilon$, then evaluating $D_{B:b}D_{A:a}(A{:}a{\cdot}B{:}b)$ and $D_{A:a}D_{B:b}(A{:}a{\cdot}B{:}b)$ would incorrectly produce different results; the latter would return $\emptyset$ because the term $A{:}a$ would be dropped when evaluating $D_{B:b}$. However, if $\delta_T X{:}a = X{:}a$ when $T \neq X$, these both correctly evaluate to ϵ.

[7] It is also necessary to simplify the results of these derivatives (e.g., $\emptyset + X = X$ or $\epsilon{\cdot}X = X$), both to ensure that DFSA construction terminates and because vestigial structure introduced in previous derivatives can substantially degrade performance (Owens et al., 2009; Might et al., 2011).

[8] Similarly, Guingne et al. (2003) lazily construct "virtual networks" in order to avoid high time/memory requirements for the evaluation of priority union, and discover an algorithm much like Brzozowski's, with their "arcset" function paralleling Brzozowski's derivative and their "finality" function paralleling Brzozowski's nullability.

[9] For the single-tape formulation, the only real need for the range of δ to be expressed as ϵ and $\emptyset$ is to enable the cleverly succinct expression of $D_c(A{\cdot}B)$ in Figure 2, where the term $\delta A{\cdot}D_cB$ is "toggled on" by ϵ being the multiplicative identity and "off" by $\emptyset$ being multiplicative annihilation. We could as easily express this toggling with boolean values and if/otherwise cases.

However, in the multi-tape formulation, the range of δ is not two-valued, and the expression of $\delta_T A{\cdot}D_{T:c}B$ as a product is necessary, since the term $\delta_T A$ may be non-trivial.

$$D_{T:c}\emptyset = \emptyset$$
$$D_{T:c}\epsilon = \emptyset$$
$$D_{T:c}X{:}a = \begin{cases} \epsilon, & \text{if } T = X \text{ and } c = a \\ \emptyset, & \text{otherwise} \end{cases}$$
$$D_{T:c}(A{\cdot}B) = D_{T:c}A{\cdot}B + \delta_T A{\cdot}D_{T:c}B$$
$$D_{T:c}(A + B) = D_{T:c}A + D_{T:c}B$$
$$D_{T:c}(A*) = \delta_T(A*){\cdot}D_{T:c}A{\cdot}A*$$

$$\delta_T\emptyset = \emptyset$$
$$\delta_T\epsilon = \epsilon$$
$$\delta_T X{:}a = \begin{cases} \emptyset, & \text{if } T = X \\ X{:}a, & \text{otherwise} \end{cases}$$
$$\delta_T(A{\cdot}B) = \delta_T A{\cdot}\delta_T B$$
$$\delta_T(A + B) = \delta_T A + \delta_T B$$
$$\delta_T(A*) = (\delta_T A)*$$

Figure 3: Brzozowski's derivative for multi-tape regular expressions. Highlighted are particularly important parts for multi-tape evaluation; when taking derivatives with respect to tape T, they serve to preserve material relevant to the other tapes.

A similar concern is seen in the formulation of $D_{T:c}(A*)$; the multi-tape formulation contains an additional term. Consider the expression $((A{:}a{\cdot}B{:}b) + A{:}x)*$; you can read this as "Any number of times, either consume an `a` on tape `A` alongside a `b` on tape `B`, or consume only `x` on tape `A` and nothing on tape `B`." Consider the derivative of this formula with respect to $B{:}b$. This will later require the consumption of the corresponding `A:a` – that is the second term $D_{T:c}A$ – but before one gets to that `a`, one could consume an infinite number of `A:x` first – this is the initial term $\delta_T(A*)$.

2.6. Additional Operators

Extending the basic objects of the system to multi-tape databases requires giving the programmer operators to handle this greater complexity, so we also implement familiar operations from relational algebra (Codd, 1983): natural join, selection, projection, and renaming.

The natural join and selection are implemented as special cases of intersection. The implementation of the rename operation (where $\rho_{T/S}L$ should be read as "rename the S tape to T in L") is as follows.

$$D_{T:c}\rho_{T/S}A = \begin{cases} \rho_{T/S}D_{S:c}A, & \text{if } S \neq T \\ \emptyset, & \text{otherwise} \end{cases} \quad (1)$$

$$\delta_T\rho_{T/S}A = \begin{cases} \rho_{T/S}\delta_S A, & \text{if } S \neq T \\ \epsilon, & \text{otherwise} \end{cases} \quad (2)$$

The inequality condition effectively says that the renamed tape S is unique – if there happens to be another tape of the same name in the grammar, and we attempt to take the derivative of this formula with respect to that tape, the derivation fails. (That is, this formula contains no content on tape S from the global point-of-view.) Renaming thus "hides" tape names from the global point of view, and we also use this property to implement projection. Tapes not included in the projection undergo Python-like "name mangling" to globally unique names so as not to clash with other names in the global namespace.

Projection is of practical importance in multi-programmer projects, because it allows encapsulation of local content. Since invoking a new tape is trivially easy (one simply starts a new column), Gramble grammars often grow to include many tapes – 10–20 is common – but many exist only to support locally-important selections (e.g., a suffix only attaches to roots of a particular class, but will not play a role in any decisions or interfaces downstream). That tape can be hidden and not form a part of that sub-grammar's "public interface", so even if one programmer names that tape `class`, it does not necessarily mean that the tape name `class` is off-limits for the rest of the project.[10]

We also implement phonological rewrite rules, roughly as in Kaplan and Kay (1994), and compose them using joins and renaming in such a way that intermediate representations are recoverable, similar to Hulden (2017). We do not, however, implement the full slate of possible rewrite rules (e.g., where rule contexts can reference material on the rule's output tape). We also have not yet implemented the lenient composition necessary for the treatment of Optimality Theory (Karttunen, 1998). For projects with complex phonology needs, a language with a richer set of phonological operations like XFST is probably still more appropriate.

3. Tabular Syntax

Gramble was originally envisioned as a "visual programming language" along the lines of Blockly

[10] For similar reasons, Gramble also supports the namespacing of identifiers. Small, academic DSLs often treat all identifiers as global, but this can lead to name clashes when projects grow from small scripts to multi-programmer codebases; e.g., one programmer defines `V` to denote the verbs and another inadvertantly re-defines it to denote the vowels.

(Fraser, 2015; Pasternak et al., 2017) for beginner programmers, Vi-xfst (Oflazer and Yilmaz, 2004) for FST development, or Galaxy-LAPPs (Ide et al., 2016) for NLP pipeline development. We had noticed beginners making frequent syntax errors in XFST/LEXC (e.g., unbalanced parentheses, missing semicolons, unescaped `0`) and thought that a "puzzle piece" interface might help mitigate these.

However, component-based visual programming typically shines for projects with perhaps 10–30 components, and the projects we had hoped to port had thousands of individual parts. Porting these to a drag-and-drop visual editor would involve creating and connecting thousands of tiny components, and result in visual tableaux that would probably be impossible to read at a glance.

In our discussions it had already become clear that the main "visual component" would need to be tabular. Most of what our SMEs deliver is in tabular form – lists of lexical roots, paradigm tables, orthographic equivalency charts, etc. – and ideally those should remain in their original forms. Also, tables naturally scale to thousands of components without hindering their readability.

This led to the thought, "If we need a tabular component, do we need anything *else*? What if everything were tabular?" Gramble thus grew into a tabular programming language akin to *Tabular* (Gordon et al., 2014), avoiding the need to bundle it with a custom editor (§3.4).

Common operations are expressed as spatial relationships between cells in such a way that the familiar columnar representation of a database maps to the multi-tape regex that would generate it. A header cell and a content cell in its column express a *tape:text* relationship, horizontal adjacency expresses concatenation, and vertical adjacency between rows expresses alternation. For a concrete example, the simple table below would compile roughly to the formula *(text:pend·gloss:love)+(text:on·gloss:see)*.[11]

text	gloss
pend	love
on	see

Special operators and headers like `replace` and `embed`, seen in the source code in Figure 1, introduce additional interpretation rules resulting in more complex formula.

We admit that tabular programming is an unconventional choice, but it ended up having many unforeseen advantages (§3.1–3.4), as well as a few disadvantages (§3.5).

3.1. Conciseness

In practice, much of the text of a multi-tape regular expression ends up being the specification of tape names. Especially when (as is often the case) each line references the same tapes line after line, columnar organization is considerably more readable. Consider the expression of the same root list in LEXC vs. Python vs. Gramble.

```
LEXICON Root                                    1
 pend@U.POS.V@@U.VAL.TR@:love                   2
     @U.POS.V@@U.VAL.TR@ NextMorph;
 on@U.POS.V@@U.VAL.TR@:see                      3
     @U.POS.V@@U.VAL.TR@ NextMorph;
```

```
root = [{                                       1
            "text": "pend",                     2
            "gloss": "love",                    3
            "pos": "v",                         4
            "val": "tr"                         5
          },                                    6
          {                                     7
            "text": "on",                       8
            "gloss": "see",                     9
            "pos": "v",                         10
            "val": "tr"                         11
          }]                                    12
```

Root =	text	gloss	pos	val
	pend	love	v	tr
	on	see	v	tr

Moving the tape names to column headers, and handling separation and hierarchy spatially, removes much of the error-prone boilerplate involved in expressing tabular information as plain text. Even if the LEXC code above were clarified with a syntax highlighter[12], the requirement that field names be specified in each flag, and (for bidirectional FSTs) that flags need to be repeated for each tape, means that each line above requires 38 added characters just to express 3 characters' worth of information.

3.2. Avoidance of Special Characters

For beginners, part of the additional complexity of programming (beyond the irreducible complexity of describing a complex system, cf. §2), is a result of the wealth of special characters and the rules for using them. For example, some special characters must be escaped to be used literally, but the escape

[11] Gramble also performs automatic detection of possible multi-character tokens, evaluating them as an atomic unit in contexts where doing so would not change the results. If we were to apply a rewrite rule to this equation, however, it would not be safe to interpret `pend` as a unit and this optimization would not apply.

[12] E.g., https://marketplace.visualstudio.com/items?itemName=eddieantonio.lexc

character is also special and thus must also be escaped; brackets must be closed (unless escaped), etc. For a non-programmer these are new ideas that must be explained before the collaborator can modify the code without breaking it.

Gramble's tabular syntax eliminates some of the common pitfalls associated with structured plain text. Expressing the gross hierarchical structure, tape naming, alternation, concatenation, and similar operations through cell relationships eliminates the need for these very frequent operations to be expressed as special characters.

With a few systematic exceptions, like # and _ having special meanings inside a rewrite rule, almost every character inside an ordinary Gramble table is interpreted literally. This is particularly important when working with linguistic glosses, which are full of special characters with special meanings (e.g., the Leipzig Glossing Rules, 2008–2015). Because of this, Gramble attempts to reserve as few special characters as possible, and have as many cells as possible default to a literal interpretation.

On the other hand, using a spreadsheet editor comes with its own set of special characters (e.g., an equals sign at the beginning of the cell indicating that it represents a formula), and working around these is admittedly rather frustrating.

3.3. Recovery from Syntax Errors

In plain-text programming languages, errors like mismatched brackets can cause an entire project not to compile, or completely change its interpretation. We have found that unhelpful error messages have been a major issue in teaching beginner programmers XFST; it can be difficult to know exactly what the error is and where it lies.

Gramble does not use bracketing for its document-level structure.[13] The interpreter does not parse strings except inside individual cells, meaning that when someone *does* make a syntax error like a mismatched bracket, it can affect only a limited region and is easy to pinpoint.

The limited scope of syntax errors also helps recover from them. For most errors, Gramble attempts to recover from them by interpreting the affected code as if it is ϵ – that is, the code will not contribute any content to resulting forms, but unlike $\emptyset$ it does not prevent them from being generated. This ability to "turn off" parts of the grammar when syntactically erroneous without compromising the ability to interpret other parts ends up being very useful in the multi-user live coding environment, since one user's syntax error will typically not have a work-stopping effect on other users.

[13]The larger-scale structure of a Gramble document is determined by a Python-like "off-side rule". This actually ends up being more ergonomic in a spreadsheet; columns are much wider than characters so it is unambiguous to which column content belongs, and there is no tab/space ambiguity.

3.4. Available Editors

When considering Gramble as a possible "visual programming language", we reviewed many examples of such, but their long-term survival prospects gave us pause. Many ended up inalienably wedded to custom development environments, and did not survive when the computing world switched to new operating systems or paradigms. Porting them to a new system meant porting their custom editors/environments, and by that point plain-text languages had already proliferated.

Spreadsheet editors, on the other hand, are ubiquitous; they will almost certainly be available on all major operating systems for the foreseeable future, and we will not have to write them ourselves. While we did write a custom plug-in (§4), we did not have to write the entire editor, let alone the sharing infrastructure. Not including the interpreter library itself, the entire plug-in is only about 1,000 lines of code.

Also, it is an interface that most knowledge workers already know intimately; we do not have to explain the intricacies of a new editor.

3.5. Downsides of Tabular Syntax

A rich ecosystem of tools has evolved to support conventional plain-text programming, and teams should consider whether the downsides of leaving that ecosystem outweigh the advantages.

In plain-text programming languages, we can use directory structure as scaffolding to organize and understand large projects. Spreadsheet editors typically lack the ability to engage with directory structure; instead, they typically use "workbooks", flat collections of worksheets with no subordinate hierarchical structure. While this flatness does help new contributors because they do not have to be taught a complex file structure, it is not ideal for a large, mature project where the possibility of such structure would help manage complexity.

It is also difficult to integrate with existing products to help manage complex multi-user projects. There is no good bridge between Google Sheets and `git`, for example; one must download the project as CSV files and manually commit them to a repository. Again, this has some advantages for the new contributor – they can use the familiar Google history feature for rollbacks rather than learn `git`, which is always a speed bump for non-programmers – but is a downside for a more experienced team that needs a richer versioning system than a history rollback.

4. Google Sheets Plug-in

We mention above that programmers transforming the SMEs' spreadsheets into working code is an irreversible process, but this is not just due to SMEs not being able to read code. It is also because this process moves the focus of development out of the SMEs' usual workspace (usually office software like Excel or Google Sheets) into the unfamiliar ecosystem of programmers.

For this reason, we have written a plug-in for Google Sheets that embeds a Gramble interpreter. Teams can add a Gramble interpreter to any sheet, which allows the system's full functionality without leaving the spreadsheet environment including:

- Syntax highlighting and the display of errors/warnings within the affected cells.
- The expression of test cases alongside grammars. The Gramble plug-in can execute these tests and highlight them green or red.
- A sidebar interface that allows the user to lookup/generate/sample forms.

Putting these abilities within the spreadsheet interface, combined with the built-in sharing and collaborative editing features, has led to a much faster iteration loop between our SMEs and programmers, since they can pair-program on a shared document.

This pair-programming helps distribute understanding system knowledge throughout the team – the SME comes to better understand the Gramble syntax and semantics, but meanwhile the programmer also comes to understand the linguistic phenomena by their closer collaboration.

An obvious concern with the use of the plug-in is that it introduces a dependency on the Google ecosystem, which raises potential privacy, trust, and longevity issues; we describe these in the ethics/limitations section following the conclusion.

5. Future Work

An important remaining question is how to optimally determine tape ordering. Although our derivative equations are specified in a way that derivatives on different tapes are commutative (i.e. $D_{A:a}D_{B:b}L = D_{B:b}D_{A:a}L$), when looking up forms or constructing a DFSA they must be queried in *some* order. Determining the optimal order for doing so is non-trivial. Although in the end the answer will be the same, it is possible to calculate derivatives in a tape order that causes the system to go down garden paths, e.g., evaluating $D_{A:x}$ repeatedly before evaluating a $D_{B:x}$ that reveals that the result is $\emptyset$.

This is made more difficult by the fact that, unlike most previous projects using multi-tape automata, we do not know the set of tapes in advance and

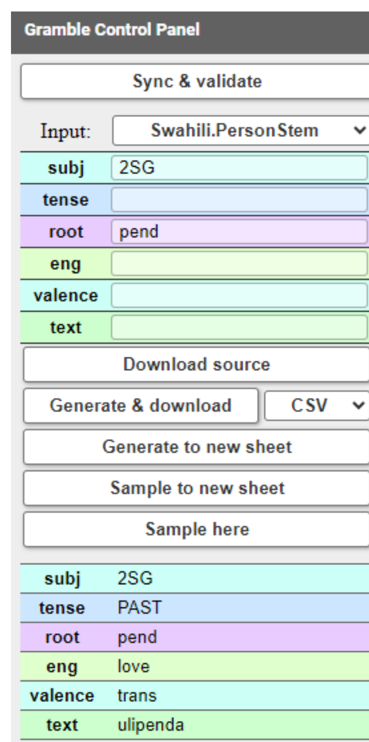

Figure 4: The Google Sheets plugin can open a sidebar allowing quick lookup/generation/sampling; here the user samples a form matching `{subj:2SG, root:pend}`.

do not necessarily know what the programmer is using them for. While we use heuristics to try to infer an optimal order, we have not managed to show that any particular order is optimal, or that an optimal order necessarily exists. (E.g., for some grammar $X+Y$, X might be optimally evaluated as $D_{A:a}D_{B:b}X$ and Y as $D_{B:b}D_{A:a}Y$.) Understanding the performance consequences of different tape-order decisions remains for future work.

Also remaining for future work is to find efficient multi-tape implementations of operators like lenient composition (§2.6), allowing Gramble to be used for more phonologically sophisticated programs.

6. Conclusion

While there is still much to be done in making Gramble more user-friendly, powerful, and efficient, we feel that it has greatly improved the quality and productivity of our community collaborations. Our collaborators have welcomed the ability to participate more directly in the programming aspects, and some have created or taken over Gramble projects without any prior programming experience.

We release Gramble under an open-source license in hope that future projects will see similar benefits, or evolve these ideas to support additional kinds of projects or modes of collaboration.

Ethical Considerations & Limitations

Language sovereignty discussions in NLP largely center around data ownership/storage/access (Keegan, 2019), and while that is obviously a crucial concern, another less-talked-about concern is the accessibility of code and the community's ability to meaningfully change it. Source code for hand-written rule-based grammars of complex languages exists in a middle ground between "data" and "code". And, as discussed in §1.1, the complexity of the programming language and ecosystem around rule-based grammars often means that simply having them *available* to a community does not necessarily mean they are *accessible*. This typically sets up a dependence on the continued availability of the original programmer, or at least their institution/lab. While the programmer/institution may be entirely willing to maintain this availability, it is still a dependency that needs to be taken into account before embarking on this kind of collaboration.[14]

The goal of this project was therefore to put collaborative dependencies in the spotlight and rethink how we approached them. In other words, we asked the question: how do the tools we use shape the types of collaborations that emerge from language modelling projects? This work attempted to address some of the technical reasons that contribute to the dynamics of division and dependence that can occur in language modelling projects. However, trying to avoid exclusion is not the same thing as actively seeking inclusion, and simply using Gramble should not be understood as a sole approach to mitigating the myriad causes that NLP collaborations can be harmful and disempowering (cf. Brinklow et al., 2019; Leonard, 2021; Bird, 2022).

3rd party data privacy As discussed in §4, our implementation is somewhat dependent on third party tools which could have varying safeguards with respect to data privacy and access. Also, it creates a dependency on another third-party; e.g., to trust that Google will continue to make Sheets available, and not make major API changes that render the plug-in inoperable.

To mitigate this, Gramble itself has been designed to avoid dependence on Google systems. The interpreter compiles to vanilla JavaScript with no framework dependencies, and can be run in any browser, or on the command line using `node`. Gramble source documents are ordinary comma-separated value (CSV) files and do not use any proprietary file formats; they could even be written in a plain text editor if the programmer prefers.

[14]See https://fpcc.ca/resource/check-before-you-tech/ for a check list to consider before adopting any technology.

Moreover, as previously mentioned, the plug-in itself is small; including both the server-side code and the GUI presented to the user, it is only 1,000 lines of code. If Google Sheets were to disappear or restrict API access, it would probably not be particularly onerous to port it to another editor that provides similar functionality.

Accessibility One of the benefits of the tabular nature of Gramble is a variety of spreadsheet editors already exist that have built-in considerations surrounding accessibility. Both Google Sheets and Microsoft Excel come with accessibility checker tools that, for example, analyze spreadsheets with respect to contrast ratio, missing titles, among other considerations, and provide services like screen reading to users.

Acknowledgments

This work would not have been possible without extensive feedback and encouragement from our community collaborators, including but not limited to Onkwawenna Kentyohkwa, the Prairies to Woodlands Indigenous Language Revitalization Circle, the Oneida Nation of the Thames, the W̲SÁNEĆ School Board, and the Kitigan Zibi Cultural Education Center. Immeasurable thanks also goes to Eddie Antonio Santos, Anna Kazantseva, Skylar Maguire, Kendra Hicks, Delaney Lothian, Akwiratékha' Martin, Yanfei Lu, and Michael Running Wolf, whose feedback, advice, and testing helped shape Gramble into its current form. Funding for this project was provided by the National Research Council Ideation Small Teams grant *Speech Generation for Indigenous Language Education*.

7. Bibliographical References

Valentin Antimirov. 1995. Partial derivatives of regular expressions and finite automata constructions. In *STACS 95*, pages 455–466, Berlin, Heidelberg. Springer Berlin Heidelberg.

Antti Arppe, Jordan Lachler, Lene Antonsen, Trond Trosterud, and Sjur N. Moshagen. 2016. Basic language resource kits for endangered languages: A case study of Plains Cree. In *Proceedings of the 2016 CCURL Workshop. Collaboration and Computing for Under-Resourced Languages: Towards an Alliance for Digital Language Diversity, LREC 2016, May 23, 2016*, pages 1–9.

Samira Attou, Ludovic Mignot, and Djelloul Ziadi. 2021. Bottom-up derivatives of tree expressions. *CoRR*, abs/2107.13373.

Kenneth R. Beesley. 2012. Kleene, a free and open-source language for finite-state programming. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 50–54, Donostia–San Sebastián. Association for Computational Linguistics.

Kenneth R Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications.

Jon Bentley. 1986. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721.

Steven Bird. 2022. Local languages, third spaces, and other high-resource scenarios. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7817–7829.

Sonja E. Bosch and Laurette Pretorius. 2006. A finite-state approach to linguistic constraints in Zulu morphological analysis. In *Studia Orientalia Electronica*, pages 205–228.

Nathan Thanyehténhas Brinklow, Patrick Littell, Delaney Lothian, Aidan Pine, and Heather Souter. 2019. Indigenous language technologies & language reclamation in Canada. *Proceedings of the 1st International Conference on Language Technologies for All*, pages 402–406.

Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM*, 11(4):481–494.

Edgar F. Codd. 1983. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69.

Fineen Davis, Eddie Antonio Santos, and Heather Souter. 2021. On the computational modelling of Michif verbal morphology. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 2631–2636, Online. Association for Computational Linguistics.

C. C. Elgot and J. E. Mezei. 1965. On relations defined by generalized finite automata. *IBM J. Res. Dev.*, 9(1):47–68.

Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop*, pages 49–50.

Andy Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgström, and John Guiver. 2014. Tabular: A schema-driven probabilistic programming language. In *POPL '14 Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 321–334. ACM Press.

Franck Guingne, Florent Nicart, Jean-Marc Champarnaud, Lauri Karttunen, Tamás Gaál, and André Kempe. 2003. Virtual operations on virtual networks: The priority union. *International Journal of Foundations of Computer Science*, 14:1055–1070.

Nizar Habash and Owen Rambow. 2006. MAGEAD: A morphological analyzer and generator for the Arabic dialects. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 681–688, Sydney, Australia. Association for Computational Linguistics.

Scott Hanselman. 2012. Dark matter developers: The unseen 99%. https://www.hanselman.com/blog/dark-matter-developers-the-unseen-99.

Mans Hulden. 2009. Foma: a finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32, Athens, Greece. Association for Computational Linguistics.

Mans Hulden. 2017. Rewrite rule grammars with multitape automata. *J. Lang. Model.*, 5:107–130.

Nancy Ide, Keith Suderman, James Pustejovsky, Marc Verhagen, and Christopher Cieri. 2016. The Language Application Grid and Galaxy. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 457–462, Portorož, Slovenia. European Language Resources Association (ELRA).

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Lauri Karttunen. 1998. The proper treatment of optimality theory in computational linguistics. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing (FSMNLP)*.

Martin Kay. 1987. Nonconcatenative finite-state morphology. In *Proceedings of EACL 1987*.

Anna Kazantseva, Owennatekha Brian Maracle, Ronkwe'tiyóhstha Josiah Maracle, and Aidan Pine. 2018. Kawennón:nis: the wordmaker for Kanyen'kéha. In *Proc. Workshop Computational Modeling Polysynthetic Languages*, pages 53–64, Santa Fe, New Mexico, USA.

Te Taka Keegan. 2019. Issues with Māori sovereignty over Māori language data.

George Anton Kiraz. 2000. Multitiered nonlinear morphology using multitape finite automata: a case study on syriac and arabic. *Computational Linguistics*, 26(1):77–105.

Christo Kirov, Ryan Cotterell, John Sylak-Glassman, Géraldine Walther, Ekaterina Vylomova, Patrick Xia, Manaal Faruqui, Sebastian Mielke, Arya McCarthy, Sandra Kübler, David Yarowsky, Jason Eisner, and Mans Hulden. 2018. UniMorph 2.0: Universal morphology. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Kimmo Koskenniemi. 1983. *Two-level morphology: A general computational model for word-form recognition and production*. University of Helsinki, Department of General Linguistics.

Kimmo Koskenniemi. 1986. Compilation of automata from morphological two-level rules. In *Papers from the Fifth Scandinavian Conference on Computational Linguistics*.

Roland Kuhn, Fineen Davis, Alain Désilets, Eric Joanis, Anna Kazantseva, Rebecca Knowles, Patrick Littell, Delaney Lothian, Aidan Pine, Caroline Running Wolf, Eddie Santos, Darlene Stewart, Gilles Boulianne, Vishwa Gupta, Brian Maracle Owennatékha, Akwiratékha' Martin, Christopher Cox, Marie-Odile Junker, Olivia Sammons, Delasie Torkornoo, Nathan Thanyehténhas Brinklow, Sara Child, Benoît Farley, David Huggins-Daines, Daisy Rosenblum, and Heather Souter. 2020. The Indigenous languages technology project at NRC Canada: An empowerment-oriented approach to developing language software. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 5866–5878, Barcelona, Spain (Online). International Committee on Computational Linguistics.

William Lane, Atticus Harrigan, and Antti Arppe. 2022. Interactive word completion for Plains Cree. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3284–3294, Dublin, Ireland. Association for Computational Linguistics.

Wesley Y Leonard. 2021. Centering Indigenous ways of knowing in collaborative language work. *Sustaining Indigenous languages: Connecting communities, teachers, and scholars*, pages 21–33.

Patrick Littell, Eric Joanis, Aidan Pine, Marc Tessier, David Huggins Daines, and Delasie Torkornoo. 2022. ReadAlong studio: Practical zero-shot text-speech alignment for indigenous language audiobooks. In *Proceedings of the 1st Annual Meeting of the ELRA/ISCA Special Interest Group on Under-Resourced Languages*, pages 23–32, Marseille, France. European Language Resources Association.

Patrick Littell, Anna Kazantseva, Roland Kuhn, Aidan Pine, Antti Arppe, Christopher Cox, and Marie-Odile Junker. 2018a. Indigenous language technologies in Canada: Assessment, challenges, and successes. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2620–2632, Santa Fe, New Mexico, USA. Association for Computational Linguistics.

Patrick Littell, Tian Tian, Ruochen Xu, Zaid Sheikh, David Mortensen, Lori Levin, Francis Tyers, Hiroaki Hayashi, Graham Horwood, Steve Sloto, Emily Tagtow, Alan Black, Yiming Yang, Teruko Mitamura, and Eduard Hovy. 2018b. The ARIEL-CMU situation frame detection pipeline for LoReHLT16: a model translation approach. *Machine Translation*, 32(1–2):105–126.

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195.

David R. Mortensen, Siddharth Dalmia, and Patrick Littell. 2018. Epitran: Precision G2P for many languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Kemal Oflazer and Yasin Yilmaz. 2004. Vi-xfst: A Visual Regular Expression Development Environment for Xerox Finite State Tool.

Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190.

Eric Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for creating a block language with Blockly. In *2017 IEEE Blocks and Beyond Workshop*, pages 21–24.

Michael Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125.

Ehud Reiter. 2021. NLG systems must be customisable. `https://ehudreiter.com/2021/02/17/nlg-systems-must-be-customisable/`.

Lane Schwartz, Francis Tyers, Lori Levin, Christo Kirov, Patrick Littell, Chi kiu Lo, Emily

Prud'hommeaux, Hyunji Hayley Park, Kenneth Steimel, Rebecca Knowles, Jeffrey Micher, Lonny Strunk, Han Liu, Coleman Haley, Katherine J. Zhang, Robbie Jimmerson, Vasilisa Andriyanets, Aldrian Obaja Muis, Naoki Otani, Jong Hyuk Park, and Zhisong Zhang. 2020. Neural polysynthetic language modelling. arXiv:2005.05477.

Lonny Alaskuk Strunk. 2020. *A Finite-State Morphological Analyzer for Central Alaskan Yup'Ik*. University of Washington.

Bruce Wiebe. 1992. Modelling autosegmental phonology with multi-tape finite state transducers. M.Sc. thesis, Simon Fraser University.