Regular expressions and predicate logic in finite-state language processing

Mans HULDEN
University of Arizona

Abstract. This paper proposes an extension to the formalism of regular expressions with a form of predicate logic where quantified propositions apply to substrings. The implementation hinges crucially on the manipulation of auxiliary symbols which has been a common, though previously unsystematized practice in finite-state language processing. We also apply the notation to give alternate compilation methods for two-level grammars and various types of replacement rules found in the literature, and show that, under a certain interpretation, two-level rules and many types of replacement rules are equivalent.

Introduction

The popularity of finite-state natural language processing can probably be partly attributed to the expansion of the the idiom of regular expressions—that is, the introduction of new regular expression operators that provide increasing layers of abstraction upon simpler regular expressions in automaton and transducer construction. Instead of designing finite-state automata or transducers manually, or even using a basic set of regular expression operators, the finite-state developer now has an array of flexible operations to choose from, including two-level rules, various flavors of rewrite rules, directional longest-match rules, context restriction rules, priority union, lenient composition, etc [1,2].

Many of these operators are naturally defined in terms of simpler regular expressions—for instance, many finite-state toolkits have a separate operator with the semantics of "contains exactly one instance of a substring drawn from the language $\mathcal{L}$," $\mathcal{L}$ being an arbitrary regular language. The same idea can of course be expressed through the more cumbersome and less legible:

$$(\Sigma^* \mathcal{L} \Sigma^*) - (\Sigma^*((\Sigma^+ \mathcal{L} \Sigma^* \cap \mathcal{L} \Sigma^*) \cup (\mathcal{L} \Sigma^+ \cap \mathcal{L}))\Sigma^*) \quad (1)$$

Unlike this example, a good many of the more advanced expressions—such as contextual rewriting, or directed replacement—are very difficult to define through basic regular expressions. The common solution throughout the research and implementation of these advanced regular expression operators has been the use of "auxiliary marker symbols"—symbols that in the process of compiling complex statements are inserted into a language, constrained, and finally removed.

For instance:

- [3] use "auxiliary brackets" to develop a rule compiler for two-level rules, where a significant portion of the description of the method is devoted to complications in compiling "overlapping" restriction rules.
- [4] make extensive use of "auxiliary brackets" which are inserted, whose presence is constrained, and which are then appropriately ignored in some contexts in defining rewrite rules and two-level rules as regular relations.
- [5], [6], and [7] use various bracketing systems to define replacement, directed replacement, and parallel replacement rules.
- [8] define a context restriction operator ($\Rightarrow$), as well as a more generalized context restriction operator, through the use of a $\diamond$-symbol, whose occurrence is constrained and which is then removed.

The possible drawbacks with this method, though very expressive, include the difficulty of post-design analysis of complex constructions, as well as verification of their correctness. In light of these problems, it would be desirable to at least systematize the notation, if not to abstract the entire construction method under a new notation where the component parts would be easy to verify for semantic correctness, and which would also be easily extended to allow the description and finite-state compilation of novel expressions.

This paper presents such an abstraction of the technique of auxiliary symbol usage in designing complex regular languages and relations. We find that, if defined in an appropriate manner, a particular kind of abstraction of this technique is equivalent to a logical notation where we can assert properties of strings in a systematic way that greatly simplifies the process of defining the types of regular languages and relations pertinent to natural language processing applications.

1. Notation

When speaking of regular languages, we denote alphabets with Δ, Γ, and Σ. We also make use of the standard extended regular expression operators: union ($\mathcal{X} \cup \mathcal{Y}$), concatenation ($\mathcal{X}\mathcal{Y}$), Kleene closure ($\mathcal{X}^*$), Kleene plus ($\mathcal{X}^+$), intersection ($\mathcal{X} \cap \mathcal{Y}$), complement ($\neg\mathcal{X}$), difference ($\mathcal{X} - \mathcal{Y}$), and asynchronous language product ($\mathcal{X} \parallel \mathcal{Y}$) (also called shuffle product). We follow the convention that individual symbols are represented in lower case, e.g. a, while arbitrary regular languages are calligraphic, e.g. $\mathcal{A}$, and reserve upper case letters to denote logical propositions, e.g. $S(\mathcal{X}, \mathcal{Y})$. We also use the standard logical quantifiers and connectives $(\exists x)$, $(\forall x)$, $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$.

2. Method

We will first outline the reasoning informally and more concretely, and later introduce the notation more formally and generally.

Consider the effect of defining a language over an alphabet $\Delta = \Sigma \cup \Gamma$, where the alphabet is divided into two parts $\Sigma = \{a, b\}$ and $\Gamma = \{\bigcirc\}$ (our auxiliary alphabet), such that it contains exactly one instance of $\bigcirc$, i.e. $(\Sigma^* \bigcirc \Sigma^*)$, and then intersecting this language with a language that contains the $\bigcirc$-symbol, and finally deleting the $\bigcirc$

symbol from the language. Let us call this auxiliary symbol removal operation $\Pi(\mathcal{L})$. For example:

$$\Pi((\Sigma^* \bigcirc \Sigma^*) \cap (\Delta^* \bigcirc a\Delta^*)) \tag{2}$$

Clearly, the end result in this example is the language over Σ^* that contains at least one a, i.e. another way of saying $(\Sigma^* a \Sigma^*)$. However, laid out in this fashion, we can see that separating the regular expression into two parts has brought about two independent statements with different semantics: the first one, $(\Sigma^* \bigcirc \Sigma^*)$ simply asserts the existence of exactly one symbol $\bigcirc$, while the second, $(\Delta^* \bigcirc a \Delta^*)$ asserts that there is a $\bigcirc$-symbol which is followed by an a. Informally, the first part says "there exists exactly one position called $\bigcirc$," and the second part: "some position called $\bigcirc$ is followed by an a."

In effect what we have achieved in intersecting these two statements and deleting the $\bigcirc$-symbol is a form of variable binding—the first regular expression being equivalent to existential quantification of a position in a string, or the "existence of a substring," and the latter a proposition bound by the variable $\bigcirc$.

We can now expand the same idea, and replace the first part of the regular expression with $(\Sigma^* ⓧ \Sigma^* ⓧ \Sigma^*)$ (for the sake of clarity in notation, let us replace the $\bigcirc$ with ⓧ in the auxiliary alphabet Δ, to make clear that our auxiliary symbol says something about a letter variable which we shall call x). We are now defining the language over Δ^* that contains exactly two symbols ⓧ. The purpose of the two ⓧ-symbols is to delineate two positions in a string x, the starting and the ending position (the usefulness of this will become clear later). Let us call the combined effect of this regular expression and of removing the auxiliary symbols the *regular expression equivalent* of $(\exists x)$, i.e. $\Pi(\Sigma^* ⓧ \Sigma^* ⓧ \Sigma^*)$. In intersecting this language before removal of the auxiliary symbols with any regular language φ (that may or may not contain ⓧ) we can achieve a propositional sentence $(\exists x)(\varphi)$.

To continue with this idea: what about the possible propositions in φ? In modelling of the open statements φ, the simplest desirable proposition would be one with the semantics that a substring is a member of some language $\mathcal{L}$, i.e. $x \in \mathcal{L}$. Over Δ^* the regular expression $(\Delta^* ⓧ \mathcal{L} ⓧ \Delta^*)$ describes precisely this circumstance: "there exists a substring ⓧ$\mathcal{L}$ⓧ." Another useful statement to have would be a kind of successor-of-relationship $S(t_1, t_2)$—t_1 is immediately succeeded by t_2—where the terms could either be arbitrary languages or variables. This translates naturally to $(\Delta^* t_1 t_2 \Delta^*)$, for example, $S(x, \mathcal{A})$ would be rendered as $(\Delta^* ⓧ \Delta^* ⓧ \mathcal{A} \Delta^*)$.

Since we have seen that $(\exists x)$ in our still tentative logic over strings can be modelled by $\Pi(\Sigma^* ⓧ \Sigma^* ⓧ \Sigma^*)$, and since a universally quantified proposition $(\forall x)(\varphi)$ is equivalent to $\neg(\exists x)\neg(\varphi)$, the regular expression equivalent of a universally quantified statement is:

$$(\forall x)(\varphi) \equiv \neg\Pi((\Sigma^* ⓧ \Sigma^* ⓧ \Sigma^*) \cap \neg(\varphi)) \tag{3}$$

We are now in a position to put together a complete logical sentence. For example, the sentence:

$$(\forall x)(x \in \mathcal{A} \rightarrow S(x, \mathcal{B})) \tag{4}$$

would describe the language where every instance of a member of language $\mathcal{A}$ is immediately followed by a string from language $\mathcal{B}$. In translating the open statements to regular expressions, we make use of the conditional laws of statement logic, where $(P \rightarrow Q) \Leftrightarrow (\neg P \vee Q)$, and we find the equivalent regular expression following the above scheme:

$$\overbrace{\neg\Pi\Big((\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*)}^{(\forall x)} \quad \cap \quad \neg\big(\neg \overbrace{(\Delta^* \textcircled{x} \mathcal{A} \textcircled{x} \Delta^*)}^{x \in \mathcal{A}} \quad \cup \quad \overbrace{(\Delta^* \textcircled{x} \Delta^* \textcircled{x} \mathcal{B} \Delta^*)}^{S(x,\mathcal{B})}\big)\Big)$$

2.1. Variables

So far we have only assumed propositions quantified by a single variable x. Naturally, we will want to extend this to an arbitrary number of variables. This requires some bookkeeping on the part of the alphabets. Suppose we have a sentence:

$$(\forall x)(\exists y)(\varphi) \tag{5}$$

Now, it will not do to define $(\exists y)$ as $(\Sigma^* \textcircled{y} \Sigma^* \textcircled{y} \Sigma^*)$ for the simple reason that this precludes the existence of $\textcircled{x}$ symbols (as $\textcircled{x}$ is not a symbol of Σ). So, with several symbols, we need the ability to describe "any symbol in Δ except $\textcircled{y}$," to ensure that we allow other auxiliary symbols in the regular expression equivalent of $(\exists y)$. This is of course easy to describe as a regular language $(\Delta - \textcircled{y})$, and as a shorthand and to keep the notation clean we shall say Δ_y signifies precisely this: any symbol in the alphabet Δ except $\textcircled{y}$. Hence, a construction such as

$$(\forall x)(\exists y)(\varphi) = \neg(\exists x)\neg\big((\exists y)(\varphi)\big) \tag{6}$$

becomes:

$$\neg\Pi\Big((\Delta_x^* \textcircled{x} \Delta_x^* \textcircled{x} \Delta_x^*) \cap \neg\Pi\big((\Delta_y^* \textcircled{y} \Delta_y^* \textcircled{y} \Delta_y^*) \cap (\varphi)\big)\Big) \tag{7}$$

Until now, we have said little about the operation $\Pi(\mathcal{L})$, except that it deletes the symbols in our "variable alphabet" Γ from the language $\mathcal{L}$.[1] Again, in order to keep the notation uncluttered, we shall define $\Pi(\mathcal{L})$ as a *dynamic* operation, that also changes the alphabet Γ, shrinking it by one symbol, which is the symbol that is currently being removed. This operation is crucial for the possible language complements that need to be taken in the process of eliminating several quantifiers. In the above example (7), for instance, the innermost Π-operation deletes the symbol $\textcircled{y}$ from the language and removes the symbol $\textcircled{y}$ from Γ, leading to that the following complement is taken with respect to

[1] From a formal language perspective, this is simply a substitution $f(\Gamma) = \epsilon$, or, from an automaton perspective, a replacement of transitions containing symbols from Γ with ϵ-transitions.

an alphabet Δ (recall that $\Delta = \Gamma \cup \Sigma$) that only contains one auxiliary $\{\textcircled{x}\}$. Likewise, after the outermost Π operation, $\Delta = \Sigma$, since all auxiliaries have now been purged from the auxiliary alphabet. This operation could be described non-dynamically, but at the cost of much lengthier expressions and without contributing to the clarity of the operation.

2.2. Propositions

We are naturally not restricted to the propositions developed so far—in fact any subset of the language Δ^* is a proposition.

Since a proposition, such as $x \in \mathcal{L}$, i.e. $(\Delta^*\textcircled{x}\mathcal{L}\textcircled{x}\Delta^*)$ may contain sublanguages where no variable symbols occur—in this example $\mathcal{L}$ may be such a language—care must be taken to ensure that other variables can freely occur within the regular expression equivalent of the proposition. Hence, propositions should in general be augmented with freely interspersed symbols from Γ, our marker alphabet. The proposition $x \in \mathcal{L}$ then becomes $(\Delta^*\textcircled{x}\mathcal{L}\textcircled{x}\Delta^*) \parallel \Gamma^*$.[2]

For example, since we can now extend the notation with any proposition, we might want to define $S(t_1, t_2)$ as n-ary, instead of a two-place predicate (which will save much ink and instantiation of variables in some constructions). It is easy to see that $S(t_1, \ldots, t_n) \equiv (\Delta^* t_1 \ldots t_n \Delta^*) \parallel \Gamma^*$, where $t_i = \textcircled{x_i}\Delta^*\textcircled{x_i}$ if the term t_i is a variable, and simply $\mathcal{L}_i$ if t_i is a language constant. For example, $S(\mathcal{L}, x, \mathcal{R})$, then becomes:

$$(\Delta^*\mathcal{L}\textcircled{x}\Delta^*\textcircled{x}\mathcal{R}\Delta^*) \parallel \Gamma^* \tag{8}$$

2.3. Interim summary

We now have a construction method by which our proposed logical notation can be systematically converted to regular expressions, and hence to finite-state automata.[3] In particular, new propositions can be introduced in a fairly straightforward way, and we shall do so whenever needed in the upcoming examples. The basic construction together with basic propositions is summarized in Table 1.

We can now proceed to tackle a selection of difficult regular language problems and illustrate their solution through the notation developed here.

---

[2]This would of course be equivalent to $(\Delta^*\textcircled{x}(\mathcal{L} \parallel \Gamma^*)\textcircled{x}\Delta^*)$, which may be more efficient to compile because of less non-determinism in the intermediate results: if $\mathcal{L}$ contains no symbols from Γ, which should be the case, then allowing symbols from Γ to freely occur within strings from $\mathcal{L}$ will not introduce nondeterminism in the automaton construction. However, for the sake of generality, we will simply say that a proposition P shall be implemented as above, with symbols from Γ occurring anywhere, i.e. $P \parallel \Gamma^*$.

[3]The overall approach is somewhat similar to classical methods of converting sentences if first-order logic of one successor FO[$<$] and monadic second-order logic MSOL[S] to finite-state automata [9,10]. There are two crucial differences, however: a) classical methods employ a joint alphabet of symbols and variables represented as vectors, while we entirely divorce the variable alphabet from our symbol alphabet, and b) we treat variables semantically as denoting substrings with a beginning and an end, rather than integers representing positions in a string. This approach, we believe, confers much more conciseness in notation, and the advantage of a simple way of defining new predicates whenever necessary, as well as being compilable into automata using existing operations of finite-state toolkits. [11] also develops a logical formalism based on the classical methods mentioned above and applies it to language processing; however, this relies on a separate compiler from MSOL[S]-logic, and requires that extra predicates be defined in terms of primitive propositions, rather than allowing intermixing regular expressions and logical statements.

Logical notation		R.E. equivalent	Notes
$(\exists x)(\varphi)$	$\equiv$	$\Pi\big((\Delta_x^* ⓧ \Delta_x^* ⓧ \Delta_x^*) \cap (\varphi)\big)$	$\Delta_x = (\Delta - ⓧ)$
$(\forall x)(\varphi)$	$\equiv$	$\neg\Pi\big((\Delta_x^* ⓧ \Delta_x^* ⓧ \Delta_x^*) \cap \neg(\varphi)\big)$	
$x \in \mathcal{L}$	$\equiv$	$(\Delta^* ⓧ \mathcal{L} ⓧ \Delta^*) \parallel \Gamma^*$	
$S(t_1, \ldots, t_n)$	$\equiv$	$(\Delta^* t_1 \ldots t_n \Delta^*) \parallel \Gamma^*$	$t_i = ⓧ_i \Delta^* ⓧ_i$ if t_i is a variable x_i
$x = y$	$\equiv$	$(\Delta^*(ⓧ \parallel ⓨ)\Delta^*(ⓧ \parallel ⓨ)\Delta^*) \parallel \Gamma^*$	

Table 1. Table summarizing the logical notation and their the regular expression equivalents. We assume the alphabets Γ and Σ, where Γ is the marker alphabet that contains the variable symbols under quantification, such as ⓧ, ⓨ, etc. Collectively, the two alphabets together are denoted Δ, i.e. $\Delta = \Gamma \cup \Sigma$. The operation $\Pi(\mathcal{L})$ deletes the currently quantified variable symbol from $\mathcal{L}$, and removes it from Γ.

2.4. An example construction

Returning to the the example construction of which a standard regular expression was given in Eq. (1), that of a language that contains only one factor from some arbitrary regular language $\mathcal{L}$; in our logical notation, we could express this as:

$$(\exists x)(x \in \mathcal{L} \wedge (\forall y)(y \in \mathcal{L} \rightarrow x = y)) \tag{9}$$

Here we need a way to model the proposition $x = y$ for some variables x and y. This circumstance in captured by the language where both ⓧ and ⓨ markers share the same positions (see table 1).

Again, using the fact that $(P \rightarrow Q) \iff (\neg P \vee Q)$, and following the translation method given we get the following regular expression:

$$\Pi\big((\Delta_x^* ⓧ \Delta_x^* ⓧ \Delta_x^*) \cap (\alpha \cap \neg\Pi((\Delta_y^* ⓨ \Delta_y^* ⓨ \Delta_y^*) \cap \neg(\neg\beta \cup \gamma)))\big) \tag{10}$$

where:

$$\begin{aligned} \alpha &= (\Delta^* ⓧ \mathcal{L} ⓧ \Delta^*) \parallel ⓧ^* \\ \beta &= (\Delta^* ⓨ \mathcal{L} ⓨ \Delta^*) \parallel (ⓧ \cup ⓨ)* \\ \gamma &= (\Delta^*(ⓧ \parallel ⓨ)\Delta^*(ⓧ \parallel ⓨ)\Delta^*) \parallel \Gamma^* \end{aligned}$$

It should be noted that there is much room for optimization in this particular construction: for instance, it is obvious that the shuffle product in unnecessary in α and partly so in γ, etc., however, we represent them explicitly here to follow the construction method mechanically. In general, depending on the nature of the propositions and the formula, some steps can be optimized or omitted to avoid unwanted nondeterminism in the intermediate stages of automaton construction.

3. Applications

3.1. Context restriction

Context restriction over arbitrary regular languages is an operation discussed in [12,8]. The notation is as follows:

$$\mathcal{A} \Rightarrow \mathcal{B}_1 \_ \mathcal{C}_1, \ldots, \mathcal{B}_n \_ \mathcal{C}_n \tag{11}$$

The intended semantics is that a context restriction statement of the format above defines the language where every instance of a substring from the language $\mathcal{A}$ is surrounded by some pair $\mathcal{B}_i$ and $\mathcal{C}_i$. This language is quite challenging to capture through standard regular expression operators, as seen in the solution given in [12].

The language of context restriction translates very naturally into a logical notation: if x is a substring that is a member of language $\mathcal{A}$, then x is the successor of a string from $\mathcal{B}$ and a string from $\mathcal{C}$ is the successor of x. Employing the n-ary successor-of predicate introduced earlier, this becomes:

$$(\forall x)\Big(x \in \mathcal{A} \rightarrow \big(S(\mathcal{B}_1, x, \mathcal{C}_1) \vee \ldots \vee S(\mathcal{B}_n, x, \mathcal{C}_n)\big)\Big) \tag{12}$$

and can be translated into a regular expression and a finite automaton exactly as described above.

3.2. Two-level rules

A two level grammar defines a subset of the language Σ_f^*, where Σ_f is the set of *feasible pairs*, defined in advance. The set Σ_f^* is constrained by the use of statements involving four operators: $a : b \Rightarrow l \_ r$ (saying the symbol $a : b$ is only permitted between l and r), $a : b \Leftarrow l \_ r$ (which says a symbol a occurring between the languages l and r must be realized as b), and $a : b/ \Leftarrow l \_ r$ (saying $a : b$ is never allowed between l and r) [13]. The notation $a : b \Leftrightarrow l \_ r$ is a shorthand for the conjunction between the first two types of rules.

The feasible pairs Σ_f are also assumed to include every symbol pair occurring in some statement in the collection of grammar rules on the left-hand side.

Compiling a collection of such rules into a finite-state automaton (or transducer) is a non-trivial task, particularly for right-arrow rules with multiple contexts. However, each of the rule types are quite comfortably expressible in the logical notation proposed:

$$\begin{array}{lcl} a : b \Rightarrow l \_ r & \equiv & (\forall x)(x \in a : b \rightarrow S(l, x, r)) \\ a : b \Leftarrow l \_ r & \equiv & \neg(\exists x)(x \in a : \overline{b} \wedge S(l, x, r)) \\ a : b \,/ \Leftarrow l \_ r & \equiv & \neg(\exists x)(x \in a : b \wedge S(l, x, r)) \end{array}$$

There is the additional practice [3,2] that right-arrow rules with multiple contexts are allowed, are separated by commas, and are interpreted disjunctively: i.e. one of the contexts must hold for the symbol pair $a : b$ to be legal. For right-arrow rules, this prompts exactly the same solution as for context restriction above:

$$(\forall x)(x \in a : b \rightarrow S(l_1, x, r_1) \vee \ldots \vee S(l_n, x, r_n)) \tag{13}$$

For left-arrow rules and disjunctive multiple contexts, the logical specification is:

$$\neg(\exists x)\big(x \in a : \overline{b} \wedge (S(l_1, x, r_1) \vee \ldots \vee S(l_n, x, r_n))\big) \tag{14}$$

that is, in every context $l_i \_ r_i$, a must be realized as b.

In essence, the above is a complete compilation algorithm and logical specification for two-level rules, if translated to regular expressions through the method presented above. The collection of individual rules are assumed to be intersected with each other and the set of feasible pairs. Hence, we get that a two-level grammar $\mathcal{G}$ can be compiled as:

$$\mathcal{G} = \Sigma_f^* \cap \mathfrak{R}$$

where $\mathfrak{R}$ is the intersection of the individual rules compiled through the notation presented here.

3.3. String-to-string two-level rules vs. replacement rules

Many proposals have been put forth regarding the conversion of so-called phonological rewrite rules into finite-state transducers. This includes [4], [7], [14], [6], inter alia. We shall here consider the construction of transducers that implement "replacement rules," as defined in [2]. However, in order to simultaneously construct transducers and clearly define the semantics of these replacement rules, we shall build them as an extension to two-level rules and interpret them as two-level correspondence rules augmented with specific *conflict resolution strategies*.

3.3.1. Replacement rules

[2] define a set of replacement rules of the format:

```
A op B dir L _ R
```

where `op` is one of `->` (replace), `@->` (replace leftmost, longest-match), `@>` (leftmost shortest-match), and `dir` one of `||` (upper-side context), `//` (left context holds on lower side, right on upper), `\\` (left context holds on upper side, right on lower), `\/` (both contexts hold on lower side). Replacement `op` may also be optional by surrounding it with parentheses, e.g. `(->)`. Multiple replacement rules may also apply in parallel, which is indicated by separating the different rules with `,,`, although in this case `op` and `dir` must be identical for all parallel rules.[4]

[4] This is based on an observation from the reference implementation of these operators, xfst 2.10.9.

The proximity between a replacement rule `a -> b || l _ r` and a two-level rule $a : b \Leftrightarrow l : \_\, r$ has been noted by [5]. Also, an "optional" replacement rule `(->)` seems to correspond somewhat to two-level right-arrow ($\Rightarrow$) rules. What stands in the way of this analogy is that replacement rules are assumed to describe a relation where the arguments are *any* regular language, while the two-level paradigm is restricting symbol-to-symbol correspondences. To bring these two formalisms closer together, we shall therefore as a first step extend the two-level formalism to allow any regular languages as arguments, and see that, under a certain interpretation, the formalisms of parallel replacement rules and string-to-string two-level rules with a conflict resolution strategy describe exactly the same relations. Through this approach we also give a clear logical definition of the two using the formalism presented in this paper.

3.3.2. String-to-string two-level rules

In what follows, we shall assume the regular semantics of two-level rules, with the addition that in statements such as:

$$A : B \;\mathsf{op}\; L \_\, R$$

where *op* is one of $\Leftrightarrow$, $\Leftarrow$, $\Rightarrow$, A and B may be *any* regular language, while L and R may be a correspondence between any two regular languages, i.e. $L_1 : L_2$. For example:

$$a^+ : x \Leftrightarrow a : \Sigma^* \_\, b : \Sigma^*$$

Also, we abandon the idea of constraining a set of feasible pairs, and replace this with the notion of *feasible string pairs*, which is a subset of $(\Sigma^* \times \Sigma^*)$, which includes all single-symbol identity pairs, as well as every language pair $A : B$ used in a two-level rule, i.e. a grammar with a single rule like $a : b^+ \Leftrightarrow x : \Sigma \_\, x : \Sigma$ is a constraint over the feasible string pairs $(Id(\Sigma) \cup a \times b^+)^*$.[5]

3.3.3. Rule conflicts

As is well known for writers of two-level grammars, rules may conflict with each other in the sense that one rule blocks the application of another rule, or two rules are mutually contradictory. These are classifiable as both left-arrow conflicts and right-arrow conflicts. An example of a left-arrow conflict, is given by the pair of rules:

$$a : b \Leftarrow l \_\, r \qquad a : c \Leftarrow l \_\, r$$

Obviously, there cannot exist a string that contains l followed by an a on the lexical side, and r, where the a is realized as both b and c on the surface side. A right-arrow conflict can be illustrated by the pair of rules:

$$a : b \Rightarrow l \_\, r \qquad a : b \Rightarrow c \_\, d$$

[5] See the appendix for a suggested encoding of regular relations as a regular language at the automaton level.

Here, the two rules state that the pair $a : b$ is only allowed in two different contexts: $l \_ r$ and $c \_ d$, in effect disallowing $a : b$ everywhere.

For practical purposes, automatic conflict resolution has usually focused on two strategies: either giving automatic precedence to one of two rules, or giving precedence to the rule which is more 'specific' in its context specification (related to the notion called 'disjunctive ordering' in serial rewrite-rule phonology).

However, in generalizing two-level rules to arbitrary strings, there is the additional possibility that a rule may be in conflict with itself.[6] Consider the rule:

$$a^+ : b \Leftrightarrow \_ \tag{15}$$

i.e., a^+ may and must be realized as b everywhere. Two hypothetical string pairs that at first glance may seem feasible are:

a a	a a
b b	b 0

However, both possibilities are ruled out: in the first case aa (a member of a^+ is corresponding to bb (not a member of the language b), and in the second case the latter a (a member of a^+) is realized as 0 (not a member of b). Hence, any input involving more than one consecutive a has no feasible realization: the rule is in left-arrow conflict with itself.

Also, consider the rule:

$$a^+ : b \Leftrightarrow l : \Sigma^* \_ \tag{16}$$

and the two pairs:

l a a	l a a
l b a	l b 0

Here, the leftmost correspondence is ruled out because a substring aa corresponds to ba, while the rule says a^+ must be realized as b everywhere. Similarly for the rightmost case: the last a is a member of the language a^+, but is realized as zero, not b.

Herein lies a crucial difference between the expected behavior of replacement rules: a replacement rule

```
a+ -> b || l _
```
(17)

will accept both correspondences described. It seems a two-level formalism based on string-to-string constraints becomes less useful than symbol-to-symbol constraints because of this stringency.

[6] Which is indeed also possible in symbol-to-symbol two-level rules, but only with epenthesis rules, i.e. rules of the type $0 : a \Leftarrow l \_ r$. This and other epenthesis-related cases will be discussed in the appendix.

If we wanted to pursue the analogy between replacement rules and string-to-string two-level rules, there is also the possibility of writing replacement rules that apply in "parallel," e.g.

```
l a r -> l a l || _ „ a -> b || l _ r          (18)
```

The way the parallel rules are defined, the two correspondences:

l a r	l a r
l a l	l b r

are allowed. However, for a similar set of two-level constraints

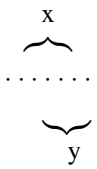
$$lar : lal \Leftrightarrow \_ \qquad\qquad a : b \Leftrightarrow l : \_\, r :$$

neither one is allowed, since they are in mutual conflict.

3.3.4. Conflict resolution

The above data suggest a strategy for automatic conflict resolution to make string-based two-level rules useable: if we would like to define transducers that *always* give some output, regardless of the input, and, at the same time, not give certain rules arbitrary precedence over others, the only feasible strategy is to have rules always yield to other rules if they are in conflict. In the previous example, we will accept both lal and lbr as outputs for the input lar, because in the two cases, rule 1 should yield to rule 2, and vice versa, mutually.

In formalizing this using our logic notation, we want to say that a rule (left-arrow or right-arrow) must hold, except if another rule permits a different correspondence in parts of the same center. To this end, we shall need a new predicate in addition to the ones already established: call this two-argument predicate OL (for 'overlaps'), with the semantics that a proposition OL(t_1, t_2) is true *iff* the positions of strings t_1 and t_2 share at least one symbol:



We can now add a statement to the previous definition of ($\Rightarrow$) to illustrate this method of conflict resolution:

$$A : B \Rightarrow L \_\, R \equiv$$
$$(\forall x)(x \in A : B \rightarrow S(L, x, R) \vee (\exists y)(OL(x, y) \wedge y \in A : B \wedge S(L, y, R))) \qquad (19)$$

that is, every $A : B$ must be surrounded by L and R, except in the case that an $A : B$ is a substring of another $A : B$ which in turn is surrounded by L and R.

However, this only solves the case where a single rule may be in conflict with itself, such as:

$$a^+ : b \Leftrightarrow \_$$

In general, we would like to enforce a right-arrow constraint only in the case that there is *no other* rule applicable for a given substring of correspondences. Assume we have a set of right-arrow rules $\{\mathfrak{R}_1, \ldots, \mathfrak{R}_n\}$ that consist of components $A_i : B_i$, L_i, R_i, and we want to express the idea that all right-arrow correspondence requirements $A_i : B_i$ must hold, except if some other correspondence (including one permitted by the rule at hand itself) is legal within the same region $A_i : B_i$.

$$\begin{aligned} &\bigwedge_{i=0}^{n} (\forall x_i)(x_i \in A_i : B_i \rightarrow S(L_i, x_i, R_i) \\ &\bigvee_{j=0}^{n} (\exists y_j)(OL(x_i, y_j) \wedge y_j \in A_j : B_j \wedge S(L_j, y_j, R_j))) \end{aligned} \tag{20}$$

There is an equivalence between replacement rules of the directional type and two-level string-to-string rules with the conflict resolution method, according to the following pattern:

`A -> B || L _ R`	$A : B \Leftrightarrow L : \Sigma^* \_ R : \Sigma^*$
`A -> B // L _ R`	$A : B \Leftrightarrow \Sigma^* : L \_ R : \Sigma^*$
`A -> B \\ L _ R`	$A : B \Leftrightarrow L : \Sigma^* \_ \Sigma^* : R$
`A -> B \/ L _ R`	$A : B \Leftrightarrow \Sigma^* : L \_ \Sigma^* : R$

However, replacement rules are somewhat more restricted since contexts are only specified on one side of a relation.

If we generalize and apply the same conflict resolution method to $\Leftarrow$, the two formalisms become equivalent. That is, the formula (20) together with an identical conflict resolution-enhanced formula for $\Leftarrow$ will produce transducers that function exactly as parallel replacement rules. Right-arrow rules by themselves produce so-called optional replacement rules `(->)`.

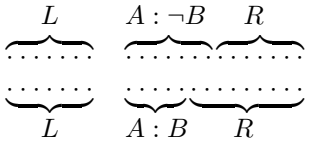
3.4. Modes of application

There is additionally the type of replacement rule that is constrained by length-of-match and direction (either left-to-right or right-to-left). So for instance, a leftmost longest-match rule like:

```
a+ @-> x
```
(21)

will map a to x, aa to x, etc [7].

Visualizing the "directionality" as either left-to-right or right-to-left conjures up a procedural image of the string correspondence, which in a logical formalism is difficult to approach. But there is a static way of looking at the formalism only in terms of string correspondences. "Longest match" in a rule that constrains a pairing of A and B (perhaps between L and R) clearly implies that a correspondence $A : B$ may not occur if that same string $A : B$ starts at the position a longer string $A : \neg B$ starts. That is, the configuration:



should be disallowed. Returning to our original definition of a right-arrow rule (disregarding for the moment what was said previously about conflict resolution), we need to restrict the right-arrow rule as follows:[7]

$$(\forall x)(x \in A : B \rightarrow S(L, x, R) \wedge \neg(\exists y)(y \in A : \neg B \wedge (x_b = y_b) \wedge (y_e > x_e) \wedge S(L, y, R))) \tag{22}$$

This defines the language where $A : B$ is only allowed between L and R, but that $A : B$ is additionally disallowed in the circumstance described above: if there also exists a substring y which starts where x starts, ends later than x and y could potentially be a legal $A : B$ correspondence, but is not.

The "leftmost" requirement is an additional constraint symmetrical to the longest-match requirement: we also want to disallow an $A : B$ whenever we can find an instance of $L\ A : \neg B\ R$ where the $A : \neg B$ portion overlaps with the position at hand and that begins earlier.

We therefore want to add the statement

$$\neg(\exists y)(y \in A : \neg B \wedge S(L, y, R) \wedge OLL(y, x)) \tag{23}$$

where OLL(y,x) is a proposition "overlaps-to-the-left," describing situations such as:[8]

$$\cdots \textcircled{y} \cdots \textcircled{x} \cdots \textcircled{y} \cdots \textcircled{x} \cdots$$

The cases of shortest-match and right-to-left directionality are entirely symmetrical: in shortest-match we disallow an $A : B$ that contains $A : \neg B$ starting at the same position, while right-to-left requires the proposition overlaps-to-the-right.

[7] We temporarily introduce two new propositions here; it is easy to see that $(x_s = y_s)$, the substring x begins where the substring y does, and $(x_e > y_e)$, the substring x ends later than y can be constructed as a regular expression over Δ—the former is the language where the first $\textcircled{x}$ is adjacent to the first $\textcircled{y}$-symbol, while the latter is the language where the second $\textcircled{x}$ occurs later than the second $\textcircled{y}$-symbol (with at least one symbol from Σ intervening).

[8] i.e. $(\Delta^* \textcircled{y} \Delta^* \textcircled{x} \Delta^* \textcircled{y} \Delta^* \textcircled{x} \Delta^*)$—this also permits the situation where the two variables denote exactly the same substring. This is a regular expression over simple strings, not correspondences. See the appendix on how to modify this to handle correspondences in our encoding.

Again, if we add the conflict resolution method outlined in the previous section to these modifications of the right-arrow rule, the semantics of the collection of replacement rules, parallel replacement rules as well as directed replacement are captured through the notation here, and are directly compilable into finite-state automata/transducers.

4. Conclusion

We have presented an extension to the formalism of regular expressions, which we call regular predicate logic. It systematizes the prevalent use of auxiliary symbols in defining complicated languages in a way that is notationally clear and can be intermixed with standard regular expressions. In particular, the propositions of our regular predicate logic are freely extendable and it is assumed that one can take advantage of other finite-state calculus operators in defining new predicates.

We have also demonstrated how the notation can be used to systematically define other formalisms used in natural language processing applications; two-level rules and replacement rules. We believe the new notation brings a level of transparency to the definition of other complex regular expression operations.

It is interesting to note that [4], in defining what they call "if-P-then-S($\mathcal{L}_1, \mathcal{L}_2$)" (the language where every string from $\mathcal{L}_1$ is followed by some string from $\mathcal{L}_2$) as $\neg(\Sigma^*\mathcal{L}_1\neg(\mathcal{L}_2\Sigma^*))$, point out an intuition that the "double complementation in the definitions ... and also in several other expressions ... constitutes an idiom for expressing universal quantification." However, in our formalism it is the *combination* of a specific type of use of auxiliary symbols together with a double negation that constitutes an idiom for universal quantification. The double negation (taken with respect to different alphabets) then becomes an artifact of the definition of universal quantification in terms of existential quantification and the construct where variable binding is achieved through intersection:

$$(\forall x)(\varphi) \equiv \neg\Pi\big((\Delta_x^* \textcircled{x} \Delta_x^* \textcircled{x} \Delta_x^*) \cap \neg(\varphi)\big)$$

4.1. Further work

The examples given in this paper are in no way meant to be exhaustive of the potential applications of the formalism. Examples of possible further work include:

- Investigation other formalisms in terms of the logical notation developed here. For instance, [15] and [1] treat Optimality Theory fundamentally through inserting violation markers, filtering them, and removing them. This can probably be interpreted within the notation at hand, where different violations are treated as different variables, which would yield a more static, logical description of OT. This would probably require an introduction of limited second-order predication together with a "cardinality" predicate (which would go beyond finite-state means in the general case), in order to keep track of the number of violations of constraints.

- Multi-tiered constraint systems could probably be described through the logical notation developed here. In essence, the string generalization of two-level grammars could probably by augmented from two to any number of tiers, where logical statements could be made within, between, and across tiers. This bears many similarities to autosegmental theories of phonology.
- The formalism itself could possibly be used as a sole basis for constructing natural language morphological analyzers, either through a 2-level or n-level representation.

5. Appendix

5.1. Multi-level encoding

In implementing string-to-string two-level rules and replacement rules, we have decided to not directly encode the relations as transducers, but rather as an even-length regular language $(\Sigma\Sigma)^*$ such that the odd numbered symbols represent the input and the even numbered symbols the output. Additionally the alphabet contains a special symbol 0 which represents ϵ (very similar to the "hard zero" in classical two-level implementations). A language over $(\Sigma\Sigma)^*$ can trivially be converted into a finite-state transducer by removing odd states in a path and creating symbol pairs from sequences, i.e. a sequence ab would become a single transition $a : b$. Also, any string pair $A : B$ mentioned in a rule is arranged in such a way that the symbol 0 is padded to the end of whichever the shorter string is to make an equal-length representation: $(abc) : d$ becomes $adb0c0$, etc. The initial "feasible language pairs" are then all doubled symbol sequences, in the example above: $(aa \cup bb \cup cc \cup dd \cup @@ \cup adb0c0)^*$, etc. The additional special symbol sequence @@ signifies an identity pair for any symbol not explicitly included in the alphabet through a left-hand side of a two-level rule.

Naturally, the logical compilation must be modified to accommodate the fact that we are dealing with symbol sequences where the symbols come in pairs: i.e. we need to also make sure the variable symbols occur at positions before even-numbered symbols from Σ so that propositions that need to refer either to the input side or the output side are consistent. Hence, $(\exists x)$ in this two-level encoding becomes:

$$((\Delta_x\Delta_x)^*ⓧⓧ(\Delta_x\Delta_x)^*ⓧⓧ(\Delta_x\Delta_x)^*)$$

and similarly for encoding the propositions.

5.2. Epenthesis rules

Epenthesis rules, exemplified in two-level grammars through rules where the left-hand side is $0 : a$, or in replacement rules such as `0 -> a` have a special status owing to the fact that the empty string ϵ, from a formal point of view, occurs an unbounded number of times within any string in a language. In fact, all approaches to treating such statements need to make explicit decisions on the semantics of epenthesis rules. Replace rules (as in [2]) come in two varieties: `0 -> L`, and `[..] -> L` with different semantics. The former is interpreted as an optional rule (inserting optionally the language L in the speci-

fied context), while the latter is an obligatory rule with the restriction that ϵ is interpreted as occurring only and exactly once between each symbol in Σ^*. These are arbitrary decisions motivated by the prevalence of epenthesis rules in phonology and the need to model such patterns through finite-state means. Their behavior can be modelled readily through the logic notation presented above.

References

[1] Dale Gerdemann and Gertjan van Noord. Approximation and exactness in finite state optimality theory. In Alain Thériault Jason Eisner, Lauri Karttunen, editor, *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, 2000.

[2] Kenneth Beesley and Lauri Karttunen. *Finite-State Morphology*. CSLI, Stanford, 2003.

[3] Lauri Karttunen, Kimmo Koskenniemi, and Ronald M. Kaplan. A compiler for two-level phonological rules. In M. Dalrymple, R. Kaplan, L. Karttunen, K. Koskenniemi, S. Shaio, and M. Wescoat, editors, *Tools for Morphological Analysis*. CSLI, Palo Alto, CA, 1987.

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

[5] Lauri Karttunen. The replace operator. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, 1997.

[6] Andre Kempe and Lauri Karttunen. Parallel replacement in finite state calculus. *Proceedings of the 16th conference on Computational linguistics*, 2:622–627, 1996.

[7] Lauri Karttunen. Directed replacement. *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 108–115, 1996.

[8] Anssi Yli-Jyrä and Kimmo Koskenniemi. Compiling contextual restrictions on strings into finite-state automata. *The Eindhoven FASTAR Days Proceedings*, 2004.

[9] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 6:66–92, 1960.

[10] C.C. Elgot. Decision problems of finite automata and related arithmetics. *Trans. Amer. Math. Soc*, 98:21–51, 1961.

[11] Nathan Vaillette. Logical specification of regular relations for NLP. *Natural Language Engineering*, 9(01):65–85, 2003.

[12] Anssi Yli-Jyrä. Describing syntax with star-free regular expressions. *11th EACL 2003, Proceedings of the Conference*, pages 379–386, 2003.

[13] Kimmo Koskenniemi. *Two-level morphology: A general computational model for word-form recognition and production*. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki, 1983.

[14] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 231–238, 1996.

[15] Lauri Karttunen. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara, 1998.