A Workbench for Logically Definable Stringsets

Dakotah J. Lambert
Earlham College
djlambe11@earlham.edu

Margaret Fero
Earlham College ('14)
mafero10@earlham.edu

Andrew Dai
Earlham College
adai13@earlham.edu

James Rogers[*]
Earlham College
jrogers@cs.earlham.edu

ABSTRACT

We have devised and implemented a suite of computational tools to support the manipulation of logically defined patterns of strings. Using these, we are building a workbench that allows one to define sets of strings in terms of systems of logical constraints, to explore the consistency and independence of those constraints and to identify the set of strings that distinguish one set of constraints from another.

Our primary application domain is the patterns of stressed syllables that occur in the words of spoken languages. The workbench provides a fully declarative and relatively transparent method of formalizing these patterns as they are described by field linguists. Because there are well-known results relating the type of the logical formulae to the language-theoretic and cognitive complexity of the sets of strings they define, the workbench can be used to establish the complexity of these constraints.

1. INTRODUCTION

Our group has been using formal logic to study aspects of phonotactics, the sound patterns of the words in human languages. Using this approach we have been able to establish a catalog of primitive logical constraints that suffice to define the patterns of phonological stress characteristic of a very large fragment of the known human languages. These primitive constraints can be combined to provide axiomatic definitions of stress patterns that can be used in a wide variety of ways: to characterize new or hypothetical stress patterns, to identify gaps and inconsistencies in the linguistic descriptions of the patterns, to explore the similarities and differences between stress patterns of individual languages, to classify languages in terms of the types of constraints, etc. Less obviously, they can be used to characterize the language-theoretic complexity of the patterns and to identify the types of cognitive resources that are required to recognize them.

[*]Faculty Advisor

Axiomatic definitions of this sort have the advantage of being declarative, unambiguous, fully explicit and generally intuitively transparent. They do, on the other hand, require a certain amount of mathematical sophistication to use. In the work reported here, we have been developing a suite of computational tools that provide a workbench to support manipulation of logically defined patterns and, ultimately, to support working with them in ways that are reasonably accessible to working phonologists.

In the next section, we sketch the theory underlying the model-theoretic definition of sets of strings. In Section 3, we explain how we have been using this to study stress patterns. In Section 4, we describe the logical and automata-theoretic machinery of the tools and in Section 5, we discuss the implementation of that machinery. We describe the minimalistic user interface of the current state of development in Section 6, and in Section 7, we consider some of the performance considerations that arise in the implementation. We close with a brief discussion of our continuing work.

2. MODEL-THEORY OF SUB-REGULAR STRINGSETS

Model theory is a way of providing precise mathematical semantics for logical formulae. The circumstances in which a formula may be true or false are modeled by mathematical structures that include a domain (or universe)—an arbitrary set—along with functions and relations on that domain that model the predicates of the logical formulae.

This provides a way of reasoning about the correctness of a set of axioms for some aspect of mathematics (number theory or set theory or geometry, for example) in fully explicit mathematical terms. It also allows one to explore the properties of the sets of models that make a particular set of formulae true and to identify properties that are common to structures occurring in very different mathematical contexts.

Conversely, it allows one to explore the expressive power of various logical languages: the limits on the ability of first-order logic, for example, to distinguish one structure from another or the kinds of predicates (less-than, for example, in contrast to successor) that are needed make those distinctions within a particular logical language. These kinds of results fall into the category of *descriptive* complexity—the complexity of the logical machinery that is needed to

	Local (+1)	Mixed $(+1, <)$	Piecewise $(<)$
MSO	Regular (Reg)		
FO	Locally Threshold Testable (LTT)	Star-Free (SF)	
Prop	Locally Testable (LT)	LT+SF	Piecewise Testable (PT)
Restricted	Strictly Local (SL)	SL+SP	Strictly Piecewise (SP)

Table 1: The Sub-Regular Hierarchy. MSO is Monadic Second-Order logic. FO is First-Order logic. Prop is Propositional logic. Restricted is Propositional Logic restricted to conjunctions of negative literals. Local is defined in terms of adjacency (+1) or, equivalently, contiguous substrings. Piecewise is defined in terms of precedence (<) or, equivalently, subsequences. The mixed classes are conjunctions of Local and Piecewise constraints.

describe a particular property of mathematical structures.

Strikingly, it turns out that there is a very close relationship between descriptive complexity of this sort and language-theoretic complexity in terms of automata and formal grammars.

We can model strings as labeled finite initial segments of the natural numbers, that is, as a finite set of positions, along with relations for successor and/or less-than and a set of positions for each symbol in the alphabet identifying the positions in which that symbol occurs.

In these terms, a set of strings is regular (a regular language) if and only if it is definable in Monadic Second-Order logic [8, 2, 4] (logic in which one can quantify over subsets as well as individuals of the domain). If one is limited to First-Order logic (quantification only over individuals) the sets of strings that are definable using less-than are Star-Free [7] (definable with regular expressions that may include complement but not Kleene-$*$), and those definable using only successor are Locally Threshold Testable [11] (definable in terms of the number occurrences of particular blocks of symbols, counting only up to a fixed threshold).

If one is limited to only Propositional fragments of logic there is no way to reason about the specific positions of symbols in a string, rather one is limited to reasoning only in terms of whether or not they occur. On this level, the distinction between reasoning with successor and reasoning with less-than becomes a distinction between reasoning about substrings (contiguous blocks of symbols) and subsequences (sequences of symbols that occur in order, but not necessarily contiguously). Table 1 provides an overview of this hierarchy of complexity classes based on their descriptive complexity.

These classes form a proper hierarchy; each class is a strict subclass of the classes above it in the table. On the bottom two levels, there is a second complexity dimension which reflects the length of the substrings or subsequences that are needed to state a constraint. Again, this forms a proper hierarchy.

From a cognitive perspective, the most useful aspect of these hierarchies is that each of the classes is characterized by the type of information about a string that a mechanism needs to be able to distinguish in order to enforce a constraint in that class [10]. To be able determine whether a string satisfies an $\mathbf{SL}_2$ constraint (strictly local with a block size of two) one needs only to be sensitive to the contiguous blocks of length two that occur in the string. To be able to determine whether a string satisfies an $\mathbf{LT}_2$ constraint, one needs to be sensitive to the entire set of contiguous blocks of length two that occur in it. Thus the relative cognitive capabilities required (by *any* mechanism) to enforce a constraint can be determined by its relative descriptive complexity.

3. AN APPLICATION DOMAIN

For the last few years our group have been using this model-theoretic approach to study the complexity of the characteristic patterns of phonological (metrical) stress that occur in natural languages.

For example, the English words “phonology” and “phonological” share roughly the same three initial syllables, but those syllables are pronounced differently. “Phonology” is pronounced with strong (primary) stress on the second syllable and weak (secondary) stress on the first, while “phonological” is pronounced with primary stress on the third syllable and secondary stress on the first. Stress of this sort can also partly determine the semantics of a word. In English the noun “record” is pronounced with primary stress on the initial syllable while in the verb “record” it is on the final syllable. One does not have to hear these in context to determine which part of speech is intended.

Each language has specific rules that govern how stress is distributed in its words. Together, those rules form a system of constraints that restrict the pronunciation of those words. This is the metrical stress pattern of the language.[1]

Our approach is to model the phonological words in a language as strings of symbols representing syllables along with

[1]This shouldn't be confused with variations in stress on the words in a sentence. Sentential stress is used, typically, to identify the topic of an utterance. Contrast, for example:
I *did not take the test yesterday*
I did ***not*** *take the test yesterday*
I did not take the test ***yesterday***.

their stress. Standardly, the only distinction between syllables that is relevant to the distribution of stress is *syllable weight*. Each language distinguishes weight in its own way, but generally there are no more than three categories: *light* (L), *heavy* (H) or *superheavy* (S). Syllables of each type may have primary stress, secondary stress or no stress. Our alphabet has a single symbol for each of these combinations. A stress pattern corresponds to a set of strings over this alphabet. Note that by working in terms of syllable weight one can compare the stress patterns of languages that have very different inventories of sounds (phonemes).

We translate those linguistic constraints into logical formulae, getting, for each stress pattern, a set of axioms that define it model-theoretically as a set of strings. The minimal syntactic complexity of those formulae corresponds to the language-theoretic complexity of the pattern and characterizes the kinds of cognitive resources an organism must have in order to detect that pattern. See [10] for a full explanation.

Using a combination of automated [3, 1, 9] and hand techniques our group has analyzed the entirety of Jeffrey Heinz's (U. Delaware) catalog of stress patterns [6] which covers a large percentage of known human languages. In prior work [12] we have factored the constraints in this catalog into the co-occurrence of *Primitive Constraints*, simple constraints of easily established complexity. Every one of the stress patterns can be axiomatized by selecting some set of these primitive constraints. The complexity of the original constraint is just the maximum of its primitive factors.

The 402 stress patterns in the database, most of which are shared by multiple languages, factor into 144 primitive constraints, most of which are shared by several patterns. These 144 primitive constraints, in turn, group into 18 categories of *Abstract Constraints* [5]. Approximately two-thirds of the stress patterns are SL. With a couple of exceptions, the rest are either SF or properly Regular when characterized locally. From the Piecewise perspective, at least eighteen are PT. (The SL constraints have not yet been analyzed with respect to the Piecewise Hierarchy.) More strikingly, very nearly all of the patterns are either SL+PT or LT+SP, with the four exceptions being a result of the same primitive constraint: that some block of syllables have even length.

4. METHODOLOGY

4.1 Logical Formulae as Automata

At the lowest level of complexity, the atomic logical formulae are either contiguous blocks of k symbols, which we refer to as k-factors, or subsequences of k symbols, which we refer to as k-subsequences. More complex formulae are built from Boolean combinations of these atomic formulae. We describe formulae of only k-factors as *local*, while those of only k-subsequences are *piecewise*. This distinction between local and piecewise formulae is a natural one from the formal perspective, as it corresponds to the distinction between successor and less-than.

It also turns out to be significant in describing phonotactic patterns. For example, the implicit constraint that all words contain no more than one primary stress can be represented with local formulae only at a high level (LTT) of the sub-Regular Hierarchy, while with piecewise formulae it is a much simpler constraint: No primary stress follows another primary stress. This rephrasal may not seem important, but it allows us to use a piecewise formula to represent the constraint at the lowest (SP) level of the hierarchy.

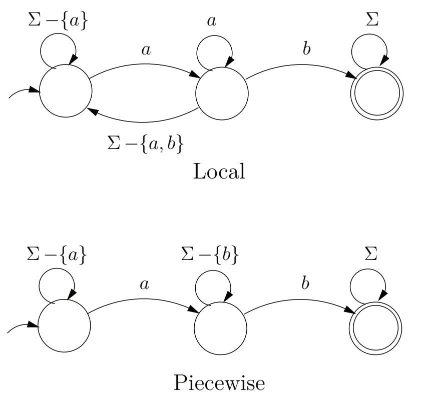

Figure 1: Automata for *ab* as a Local and a Piecewise formula

Both local and piecewise formulae, and thus constraints based on them, can be represented as finite-state automata. Examples are given in Figure 1. The set of strings accepted by the Local automaton is exactly the set of strings that satisfy the atomic formula '*ab*' in the local sense, which is the set of strings that include '*ab*' as a *substring*:

$$\{w_1 a b w_2 \mid w_1, w_2 \in \Sigma^*\}.$$

The set of strings accepted by the Piecewise automaton is the set of strings that satisfy '*ab*' in the piecewise sense, the set of those that include '*ab*' as a *subsequence*:

$$\{w_1 a w_2 b w_3 \mid w_1, w_2, w_3 \in \Sigma^*\}.$$

The set of strings that satisfy the logical conjunction of two formulae is the intersection of the sets that satisfy the individual conjuncts. The corresponding automaton can be obtained using a standard automaton construction for intersection. Similarly, automata for the logical disjunction of two formulae can be obtained using a standard construction for union. In our implementation, both intersection and union are realized via the product construction.

The set of strings that satisfy the negation of a formula is the absolute complement of the set that satisfy that formula. In automata theoretic terms, this can be obtained simply by reversing the roles of accepting and non-accepting states, but only if the automaton is *deterministic*—if for every string $w \in \Sigma^*$ there is exactly one computation of the automaton in w.

The linear time algorithm for deciding if a given string is accepted by an automaton also depends on the automaton being deterministic. But some operations on finite-state automata are most easily implemented in a way that results in a non-deterministic result. Determinizing an automaton results in exponential increase in the automaton's size, as it requires building a transition relation from each subset of its states by a symbol to the set of states reachable. This is the powerset construction. It is a theorem that this exponential blow-up is, in general, unavoidable; logical formulae are an exceptionally concise means of defining finite state automata. The challenge is to avoid blowing up the size of automata at intermediate stages of a construction only to simplify them again later.

4.2 Semantic Properties

Satisfaction is the most basic semantic property of formal languages. If a string meets the conditions a formula describes, then it is said to *satisfy* that formula. Moreover, a string can satisfy a set of formulae, Φ, by satisfying each formula $\varphi \in \Phi$. In order to determine whether a string satisfies a formula, φ, we apply the recognition algorithm on the automaton representing φ.

Consistency of a set of formulae is satisfiability of that same set. A set of formulae Φ is consistent if and only if there is some string that satisfies Φ. This is true whenever the conjunction of the formulae is not the empty set.

$$\bigwedge \Phi \neq \emptyset.$$

We use this definition when testing for consistency with the workbench. An automaton representing Φ is constructed, then we apply the emptiness algorithm to this automaton. The emptiness algorithm requires testing this automaton for isomorphism to the automaton representing the empty language.

Further, logical consequence is defined in terms of consistency. A formula ψ is a logical consequence of Φ if and only if the set $\Phi \cup \{\psi\}$ is inconsistent. In our workbench, we test for this using the complement construction and the emptiness algorithm:

$$\left(\neg\psi\right) \wedge (\Phi) = \emptyset.$$

Finally, independence of formulae is defined in terms of logical consequence. In order to find a set of independent formulae from a set Ψ, we iterate over the formulae and use the logical consequence algorithm, where ψ is bound to the current formula, and Φ is bound to the remainder of the set. If ψ is not a logical consequence of Φ, then it is independent.

5. IMPLEMENTATION

We use Haskell[2], a functional programming language, to define the data types and algorithms used in the workbench. We believe this choice of programming language facilitated development of this workbench, because the functional style and inbuilt set operations are conducive to specifying logical or mathematical descriptions of problems. Other functional languages share these properties, but we had prior experience with Haskell.

In this workbench we represent automata as four-tuples of the form (Σ, δ, I, F), where Σ represents the alphabet of the automaton, δ is a representation of the transition relation, I is the set of possible initial states, and F is the set of accepting states. For the sake of computational efficiency, each automaton also has a Boolean flag indicating whether the automaton is deterministic. The transition relation specifies a state of departure, a symbol from Σ, and a state of arrival. The transition relation is only functional when the automaton is deterministic. A state is simply an object specified by a label. To allow the workbench to generate graphs from the automaton-representation of a language, we have made considerations for including LaTeX fragments within the generated graph. Namely, a symbol is specified in our workbench by a string rather than a single character.

A string is a sequence of symbols, which is encoded in our workbench as a standard Haskell list containing symbols. To test whether a string s is contained in a language L, each path that s traces along the edges of the graph of L is generated. The string s is in L if and only if one of these paths ends in an accepting state.

Like the problem of acceptance, many operations on finite state automata reduce to generating sets while traversing graphs. For example, the union or intersection of two languages, L_1 and L_2, can be found by first generating graphs, $\mathcal{M}_1$ and $\mathcal{M}_2$ that accept all and only the strings in their respective languages. Beginning in an initial state of each graph, construct a set of transitions $\{(q_1 \times q_2, \sigma, q_1' \times q_2')\}$ where each of q_1, q_1' are states in $\mathcal{M}_1$, q_2 and q_2' are states in $\mathcal{M}_2$, and both (q_1, σ, q_1') and (q_2, σ, q_2') are transitions in their respective graphs. The alphabet of the new automaton is simply the union of those of the inputs. The set of initial states is the set $\{q_1 \times q_2 \colon q_1 \in I_1, q_2 \in I_2\}$. The only aspect in which the intersection and union of two automata differ with regard to this algorithm is which states are accepting. For the case of intersection, F is the set $\{q_1 \times q_2 \colon q_1 \in F_1, q_2 \in F_2\}$. However, the union relaxes this condition, requiring only one of q_1 and q_2 to be accepting states in the original automata.

Some operations require no traversals at all; finding the complement of a language requires only computing $Q \setminus F$ where Q is the set of all states in the graph generated by the language. Other operations are much more complicated, such as constructing a minimal deterministic automaton for a given language. In order to do this, first the automaton must be determinized if it is not already.

Determinization requires first removing all edges on ε. This is done by, for each state q, adding an outbound transition to any state that can be reached by any number of ε-transitions from the state of arrival of any edge departing from q. After these new edges have been generated, the original ε-edges are removed. The automaton is the converted to a deterministic one via the powerset construction. We then use Edward F. Moore's algorithm to minimize the resulting deterministic automaton.

To check for isomorphism of two automata, $\mathcal{M}_1$ and $\mathcal{M}_2$, the differences $\mathcal{M}_1 \setminus \mathcal{M}_2$ and $\mathcal{M}_2 \setminus \mathcal{M}_1$ are computed. If and only if both are equal to $\emptyset$, then the automata are isomorphic. In our workbench, the equality operator is defined as a check for isomorphism, as that represents equality between languages. Concatenation and the Kleene star are also implemented, using the standard non-deterministic approach, resulting in a complete set of operations on automata. Ev-

[2]www.haskell.org/haskellwiki/Haskell

ery other operation can be specified in terms of the defined operations.

In fact, having both union and intersection is redundant, as one can be defined from the other and complement. However, implementing both directly results in a slightly more efficient implementation, at the cost of some code duplication. We provide two functions in the workbench to allow users to define automata without relying on a particular internal format. These are `singleton` *`string`* and `empty`, which return the automata associated with a language containing only a single string and that containing nothing, respectively. A function `complete` *`symbols fsa`* exists to extend the alphabet of a given automaton to include the specified symbols.

6. A MINIMAL USER INTERFACE

The primary interface to the workbench is *GHCi*[3] coupled with an external module defining additional functions on automata. Optionally, a parser may be included to generate automata from regular expressions. The syntax of the workbench allows operations to be specified in words or symbolically, e.g. ``x `implies` y`` is equivalent to `x ==> y`. The available symbolic alternatives are summarized in Table 2.

`/\\`	Intersection
`\\/`	Union
`==>`	Implication
`<==`	Reverse implication
`<==>`	Bidirectional implication
`<>`	Concatenation
`==`	Equality
`/=`	Inequality

Table 2: Symbolic operators.

At the time of this writing, there are no symbolic alternatives for unary operations like finding the complement of a language, or the result of the Kleene star operator. Additionally, the functions used to test for emptiness or acceptance, as well as those used to create automata, are without symbolic representations. Appendix A provides an example of how one provides a language to the workbench.

7. PERFORMANCE ISSUES

After performing a few operations on an automaton, the number of states in the machine increases quickly. Each intersection or union of automata $\mathcal{M}_1$ and $\mathcal{M}_2$ results in an automaton with as many states as the product of the number of states in $\mathcal{M}_1$ with that of $\mathcal{M}_2$. Determinization results in a potentially exponential growth in the number of states.

This large growth rate is inhibited by multiple methods. First, there is a flag associated with each automaton describing whether or not it is deterministic. This prevents unnecessary determinization. Additionally, the states generated in the product and powerset constructions are all and only those that are visited by the input automata. Finally, a threshold is set where every automaton with more states than the threshold automatically undergoes the minimization algorithm.

This threshold, it turns out, is quite important. Testing the workbench using both Intel and PowerPC architectures on multiple sample languages of differing complexities revealed that performance is best when the threshold is set to around 10, with degradation of more than one hundred percent when the threshold is adjusted to 15. Further testing may reveal ways to enhance our estimate of an ideal threshold, or perhaps allow us to implement a dynamic threshold.

Another aspect of the workbench that hindered run time was our use of lists rather than sets to represent the components of our automata. For the current state of our implementation, nearly every operation involving a Haskell list comprehension against components of the automata was changed to a "filter, map, reduce" formula that increased both the maintainability (fewer lines of code) and the computational efficiency (fewer instructions) of the workbench.

8. FUTURE WORK

We hope to reach feature parity with *foma*[4], another tool used to interact with finite-state automata and transducers. After reaching a minimal level of feature parity here, we hope to support characterizations by First-Order and Monadic Second-Order logic. Additionally, we hope that our workbench can be used to perform field-analysis both of languages and of other similarly characterized patterns.

9. REFERENCES

[1] G. Bailey, M. Edlefsen, M. Visscher, D. Wellcome, and S. Wibel. Deciding strictly piecewise stringsets. In *Proceedings of the Midstates Conference for Undergraduate Research in Computer Science and Mathematics (MCURCSM'09)*, 2009.

[2] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[3] M. Edlefsen, D. Leeman, N. Myers, N. Smith, M. Visscher, and D. Wellcome. Deciding strictly local (SL) languages. In J. Breitenbucher, editor, *Proceedings of the Midstates Conference for Undergraduate Research in Computer Science and Mathematics (MCURCSM'08)*, pages 66–73, 2008.

[4] C. C. Elgot. Decision problems of finite automata and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–51, 1961.

[5] M. Fero, D. Lambert, S. Wibel, and J. Rogers. Abstract categories of phonotactic constraints. In *National Conference on Undergraduate Research*, 2014.

[6] J. Heinz. UD phonology lab stress pattern database. http://phonology.cogsci.udel.edu/dbs/stress/, 2012.

[7] R. McNaughton and S. Papert. *Counter-Free Automata.* MIT Press, Cambridge, MA, 1971.

[8] Y. T. Medvedev. On the class of events representable in a finite automaton. In E. F. Moore, editor,

[3]The interactive environment supplied by the Glasgow Haskell Compiler. www.haskell.org/ghc

[4]code.google.com/p/foma

Sequential Machines—Selected Papers, pages 215–227. Addison-Wesley, 1964. Originally in Russian in Avtomaty (1956), pp. 385–401.

[9] J. Rogers, J. Heinz, G. Bailey, M. Edlefsen, M. Visscher, D. Wellcome, and S. Wibel. On languages piecewise testable in the strict sense. In C. Ebert, G. Jäger, and J. Michaelis, editors, *The Mathematics of Language: Revised Selected Papers from the 10th and 11th Biennial Conference on the Mathematics of Language*, volume 6149 of *LNCS/LNAI*, pages 255–265. FoLLI/Springer, 2010.

[10] J. Rogers, J. Heinz, M. Fero, J. Hurst, D. Lambert, and S. Wibel. Cognitive and sub-regular complexity. In G. Morrill and M.-J. Nederhof, editors, *Formal Grammar 2012*, volume 8036 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2012.

[11] W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and Systems Sciences*, 25:360–376, 1982.

[12] S. Wibel, M. Fero, J. Hurst, D. Lambert, and J. Rogers. Classifying relative complexity of factored stress patterns. In J. Heinz, H. van der Hulst, and R. Goedemans, editors, *Univ. of Delaware Conference on Stress and Accent*, 2012.

APPENDIX

A. CAMBODIAN AND THE WORKBENCH

Cambodian, as described in [6], has the following constraints:

- Primary stress falls on the final syllable,
- Secondary stress falls on all heavy syllables,
- Light syllables only occur immediately following heavy syllables, and
- Light monosyllables do not occur.

Additionally, it is assumed that every language has the constraint that exactly one syllable of primary stress occur in any word. Given these constraints, we define sets of logical formulae to represent each constraint. What follows is our characterization of the language.

In order to require a syllable of primary stress to occur at the end of a word, there are actually four constraints that must be met:

	No final unstressed syllables
$\neg\,\sigma\,\ltimes$	`noUssEnd = complement ((star xss) <> uss)`
	No final secondary-stressed syllables
$\neg\,\grave{\sigma}\,\ltimes$	`noSssEnd = complement ((star xss) <> uss)`
	Nothing follows primary-stressed syllables
$\neg\,\acute{\sigma}\,\overset{*}{\sigma}$	`noPssXss = complement (pss <> xss)`
	No empty words
$\neg\,\rtimes\,\ltimes$	`noEmpty = contains xss`

The requirement that all heavy syllables have (at least) secondary stress is simpler, requiring only what it claims to:

	No unstressed heavy syllables
$\neg\,H$	`noUsH = complement (ush)`

Requiring that light syllables occur only immediately after a heavy syllable is a combination of two constraints:

	No two consecutive light syllables
$\neg\,\overset{*}{L}\,\overset{*}{L}$	`noXsLXsL = complement (xsl <> xsl)`
	No intial light syllables
$\neg\,\rtimes\,\overset{*}{L}$	`noStXsL = complement (xsl <> (star xss))`

In fact, this is all that is necessary to describe Cambodian. The language is fully described by the conjuction of these constraints, as each of the others is a logical consequence of this set. Logically the conjunction is:

$$\left(\neg\,\sigma\,\ltimes\right)\wedge\left(\neg\,\grave{\sigma}\,\ltimes\right)\wedge\left(\neg\,\acute{\sigma}\,\overset{*}{\sigma}\right)\wedge\left(\neg\rtimes\ltimes\right)\wedge\left(\neg\,H\right)\wedge\left(\neg\,\overset{*}{L}\,\overset{*}{L}\right)\wedge\left(\neg\rtimes\overset{*}{L}\right)$$

while using our workbench it is:

```
cambodian =   noUssEnd /\\ noSssEnd /\\
              noPssXss /\\ noEmpty /\\
              noUsH /\\ noXsLXsL /\\ noStXsL
```