Stress Rules from Surface Forms: Experiments with Program Synthesis

Saujas Vaduguru[1] **Partho Sarthi**[2] **Monojit Choudhury**[3] **Dipti Misra Sharma**[1]
[1] IIIT Hyderabad [2] University of Wisconsin, Madison*
[3] Microsoft Research India
[1] {saujas.vaduguru@research.,dipti@}iiit.ac.in
[2] sarthi@wisc.edu
[3] monojitc@microsoft.com

Abstract

Learning linguistic generalizations from only a few examples is a challenging task. Recent work has shown that *program synthesis* – a method to learn rules from data in the form of programs in a domain-specific language – can be used to learn phonological rules in highly data-constrained settings. In this paper, we use the problem of phonological stress placement as a case to study how the design of the domain-specific language influences the generalization ability when using the same learning algorithm. We find that encoding the distinction between consonants and vowels results in much better performance, and providing syllable-level information further improves generalization. Program synthesis, thus, provides a way to investigate how access to explicit linguistic information influences what can be learnt from a small number of examples.

1 Introduction

Deep neural models have driven recent success in NLP, including models for tasks in phonology and morphology. They have been applied to tasks such as grapheme-to-phoneme conversion (Ashby et al., 2021) and morphological reinflection (Pimentel et al., 2021), and also been incorporated into theories of phonology (Wu et al., 2021). While neural models are powerful learning machines, they require a large number of training examples, either for supervised or for transfer learning. Additionally, these models are not easily interpretable, and understanding what stuctures and patterns these models learn from data is a non-trivial task.

In this paper, we explore an different approach to learning linguistic patterns from data – *program synthesis*. Program synthesis (Gulwani et al., 2017) is a method to learn interpretable rules in the form of programs in a *domain-specific language* (DSL). The design of the DSL allows for specifying domain information, such as various linguistic concepts, and using these to guide learning from data.

Word	Stress pattern
sata	0100
hiha	0100
vatova	000100
kahasi	000100
ʔaona	01000
dehiaʔhe	00001000

Table 1: An example of the task of predicting stress patterns based on surface forms from the Cofan language. Each phoneme in each word is labelled with 1 for primary stress or 0 for secondary stress.

Vaduguru et al. (2021) show that program synthesis can be used to learn linguistic rules from a small number of examples, and apply it to learning phonological rules that perform string-to-string transformations. They demonstrate their method for learning different types of rules, including morphophonology, transliteration, and phonological stress.

In this paper, we investigate how the design of the DSL influences what rules are learnt from data. To do this, we focus on learning rules that determine placement of phonological stress from data. Phonological stress depends on both the position of a syllable within words, and language-dependent syllable weight hierarchies. This allows us to study how encoding information about position within a word and distinctions relevant to syllable weight hierarchies affects a program synthesis system designed to learn these rules from only a small number of examples.

We extend the formulation of phonological stress placement as a string-to-string transformation prob-

*Work done while at the Microsoft Research India

Proceedings of the 18th International Conference on Natural Language Processing, pages 619–628
Silchar, India. December 16 - 19, 2021. ©2021 NLP Association of India (NLPAI)

lem from Vaduguru et al. (2021) and develop a program synthesis approach specific to stress. We design different DSLs, each providing access to different phonological abstractions. We compare the results from using these different DSLs on data from a variety of languages.

Through the example of using program synthesis to learn stress rules, we seek to illustrate how program synthesis can be used as a general framework to compare how providing the same learning algorithm access to different linguistic abstractions can influence generalization from some given data.

2 Program Synthesis

Program synthesis is the task of finding a program in a domain-specific language (DSL) that satisfies certain constraints (Gulwani et al., 2017). This approach allows us to encode domain-specific assumptions about a task, and use generic search-based (Polozov and Gulwani, 2015) or constraint-based (Solar-Lezama, 2008) approaches to synthesize programs.

We apply program synthesis to learn rules for stress placement. The programs that are synthesized operate directly on the surface form of a word, which is provided as a string. By varying the structure of the DSL, we control the kind of phonological abstractions that are available to the synthesizer, and observe the effects of this on learning and generalization.

2.1 Stress rules as programs

We model stress rules as string-to-string transformations. Formally, we synthesize program that implements a function $f : \Sigma^* \rightarrow \{0, 1, 2, 3\}^*$, where Σ is the set of phonemes in a language. f takes as input a sequence of phonemes $w_1 w_2 \dots w_n$, and assigns a "degree of stress" to each phoneme. 0 indicates that a phoneme is unstressed, 1 indicates primary stress, 2 secondary stress, and 3 tertiary stress. Since stress is applied at the level of the syllable, we conventionally mark the first vowel of a syllable with the degree of stress, treating it as the 'locus' of stress within a syllable. We refer to this output string composed of the degree of stress for each phoneme in the input word as the *stress pattern* for the word.

The programs we synthesize take the form of sequences of rules similar to rewrite rules (Chomsky and Halle, 1968).

Each rule is of the form

$$\phi_{-l} \cdots \phi_{-1} X_1 \cdots X_c \phi_1 \cdots \phi_r \rightarrow T \qquad (1)$$

A rule applies to a central phoneme which satisfies a conjunction of predicates $X_1, \dots, X_c$, which appears in a context defined by the conjunction of predicates $\phi_{-l}, \dots, \phi_{-1}$ (which apply to l phonemes to the left of the central phoneme) and $\phi_1, \dots, \phi_r$ (which apply to r phonemes to the right of the central phoneme). If the conjunction of all predicates is satisfied, a transformation T is applied to the phoneme.

The DSL defines the set of predicates $\mathcal{P}$ and set of transformations $\mathcal{T}$ that can be used. We vary the predicates ($\mathcal{P}$) available to the synthesizer to define different DSLs, each providing access to a different classes of phonological abstractions. The transformation is a function that takes the phoneme as input and outputs the degree of stress, and is of the form `ReplaceBy(`s`)`, where s is a value representing the degree of stress.

2.2 Domain-specific languages

A domain-specific language is a declarative language that defines the set of programs within which we need to search. It is defined by a set of operators, their semantics, and a grammar that defines rules to combine operators. Each operator also has an associated score, which can be combined with the scores for other operators in the program to derive a *ranking score* that can be used to break ties among multiple correct programs. By appropriately choosing the operators, their semantics, and the scores associated with them, we can control domain-specific knowledge and preferences available to the synthesizer.

We use a DSL that implements rules of the form described in Section 2.1 using if-then-else constructs. This allows us to define a sequence of rules, the application of which is conditioned on a conjunction of predicates. The first rule for which the condition is satisfied is executed. The sequence of rules is applied to each phoneme of the input to obtain the degree of stress on that phoneme. This is achieved using a `Map` operator.

As described in Section 2.1, the condition for the `IfThenElse` constructs is defined as a conjunction of predicates. Based on the set of predicates available to the DSL, we define a sequence of 4 DSLs, each of which provides access to a different set of phonological classes (sets of phonemes).

```
output := Map(rules, input_phonemes)
rules := IfThenElse(C, T, rules) | T
```

Figure 1: `IfThenElse` statements in the DSL. A transformation `T` is applied if the condition `C` is true, else a transformation determined by the remaining `rules` is applied.

A predicate is defined by a predicate type and a class of phonemes to which it applies. We define various classes, and use groups of these classes to define different DSLs.

2.2.1 Classes of phonemes

The most basic set of classes is the set of singleton classes, each referring to one phoneme. We then define classes of consonants and vowels. Most stress systems do not distinguish between different (short) vowels to determine syllable weight hierarchies. Allowing this distinction to be made can allow the synthesizer to learn rules that identify syllable types as a sequence of vowels and consonants in a specific order.

Phonemes that share phonological features are also grouped into classes. We include vowel features such as height and frontness, and also features of consonants such as place and manner of articulation.

Finally, we define classes based on syllable-level information, such as whether a phoneme is the first vowel of a long vowel, diphthong, open syllable, or closed syllable. For our synthesizer, we define these uniformly across languages. A diphthong refers to a sequence of two different vowels. We treat a syllable as closed when the vowel is followed by multiple consonants, and break the syllable after the first consonant. A syllable that is not closed is treated as open.

The classes that are available to each of the 4 DSLs we define – BASIC, CV, SYLLABLE, and FEATURE – are shown in Figure 2.

2.2.2 Predicate types

We define a number of predicate types, which determine the positions in the word to which the predicate applies. In each of these cases, `X` refers to a class of phonemes. We will illustrate how each of this predicate types, defined for one unit, can be used as part of a hypothetical stress rule.

`IsX` predicates determine whether a phoneme is a member of a particular class. For example, since consonants are not stressed, `IsConsonant` can be used to ensure the output at a consonant is 0.

`IsKthX` predicates take an additional argument K, and determine if a phoneme is the Kth occurence of a member of a class in the word. If a stress rule places primary stress on the second closed syllable of a word, then the `IsKthClosedSyllable` predicate can be used with $K = 2$ to select that syllable. Note that K can also be negative, to refer to units counting from the right edge of the word.

Each of these predicates may apply to either the central phoneme (one of the X_i from eq. (1)), or phonemes in the context (one of the ϕ_i from eq. (1)). We guide the synthesizer to prefer simpler rules by ranking predicates that refer to nearby phonemes (at a smaller displacement from the central phoneme) above those that refer to more distant phonemes. For `IsKthX` predicates, we rank predicates that take a smaller absolute value of K higher to guide the synthesizer to prefer rules that refer to edges of the word over arbitrary positions in between. We also define additional types of predicates that can refer only to the central phoneme.[1]

`PrefixContainsX` predicates check whether the prefix of the word up to, but not includ-

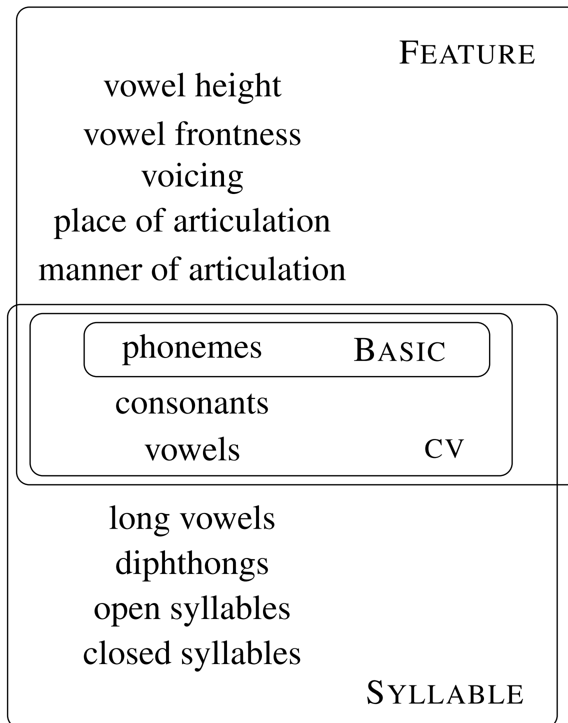

Figure 2: Classes available to each DSL – BASIC, CV, SYLLABLE, and FEATURE.

[1] These predicates are not included in the FEATURE DSL due to the requirement of enumerating a very large number of predicates.

ing, the phoneme contains any instances of a class. If a stress rule places primary stress on the first occurrence of /e/ in a word, then `PrefixContainsPhoneme(e)` can be used to ensure other occurrences of /e/ are not stressed.

SuffixContainsX predicates check whether the suffix of the word after, but not including, the phoneme contains any members of a class. Similar to the example above, if the last occurrence of /e/ is to be stressed, `PrefixContainsPhoneme(e)` can be used to ensure other occurrences are not stressed.

WordContainsX predicates check whether a member of the class exists anywhere in the word. If a stress rule places stress on the first vowel of the word if there are no long vowels in the word, then `WordContainsLongVowel` can be used to ensure stress is not placed on the first vowel incorrectly.

2.3 Synthesis algorithm

Synthesis begins with extracting phoneme-aligned pairs from the words. Each example is a pair of a phoneme and the degree of stress with which it is labelled. The synthesis algorithm then learns rules that map a phoneme (in its context) to the correct label.

To synthesize `IfThenElse` constructs, we adapt the LearnProgram, LearnBranch, and LearnConj procedures from Kini and Gulwani (2015). These procedures allow for learning *decision lists*, which are sequences of predicate-transformation pairs of the form $\langle (p_1, t_1), (p_2, t_2), \ldots, (p_n, t_n) \rangle$, where each p_i is a conjunction of atomic predicates introduced before, and t_i is a transformation function. The list is constructed such that given a set of examples X, the set can be partitioned into n subsets such that for the ith subset X_i is does not satisfy any of the predicates $p_1, \ldots, p_{i-1}$, and satisfies p_i, and the transformation t_i results in the correct output for the examples in X_i. These decision lists correspond to nested `IfThenElse` constructs. An example is tested for the predicate p_i. If the predicate is true of the example, t_i is executed, and execution is terminated. If not, the else clause – which represents the rest of the list – is executed.

The LearnProgram procedure learns a decision list given a set of input-output examples X, optimizing for a shorter list. The procedure maintains a set R of examples which haven't yet been covered by any of the predicates of the decision list, which is initialized with the entire set X. The procedure then calls LearnBranch, which learns the next element of decision list – a predicate that determines when the item will apply, and a corresponding action. Examples which satisfy the predicate are then removed the from R. This is repeated till R is empty.

LearnBranch starts by generating a set of candidate transformations. Each transformation divides the set of examples into two – those it transforms correctly, and those it does not. Then, the LearnConj can be used to obtain conjunctions of candidate atomic predicates that are true for the most examples in the former set, and false for all examples in the latter set. The conjunctions with the best ranking scores (determined as the sum of the scores for individual atomic predicates) are then each combined with the transformation to obtain predicate-transformation pairs. The predicate-transformation pair that covers the largest number of examples is then chosen as the next element of the decision list.

The LearnBranch and LearnConj procedures require the synthesis of candidate atomic predicates and transformations. These are synthesized using the FlashMeta algorithm. Given the transformation or predicate operator (as described in Section 2.2), FlashMeta can be used to infer arguments to the operator such that it satisfies a given set of examples. Based on the examples, FlashMeta finds the position of phonemes to which a predicate applies relative to the central phoneme (a value between $-l$ and r in eq. (1), where 0 refers to the central phoneme), and values of additional arguments to the predicate such as the value of K in predicates of the type `IsKthX`. FlashMeta also finds the output value for transformation operators. To do this, FlashMeta uses the *inverse semantics* of the operators, which constrains the values of arguments given the behaviour of the operator as input-output examples. Figure 3 illustrates the working of the synthesis algorithm.

3 Dataset

We obtain data by consulting grammars and other linguistic and phonological analyses of languages listed in the StressTyp2 database (Goedemans et al., 2014) or by Gordon (2002). The database contains information about various lects and the kinds of stress patterns exhibited by these lects. The database also has links to the sources from which the data was collected for compiling the database, and these were the sources we consulted

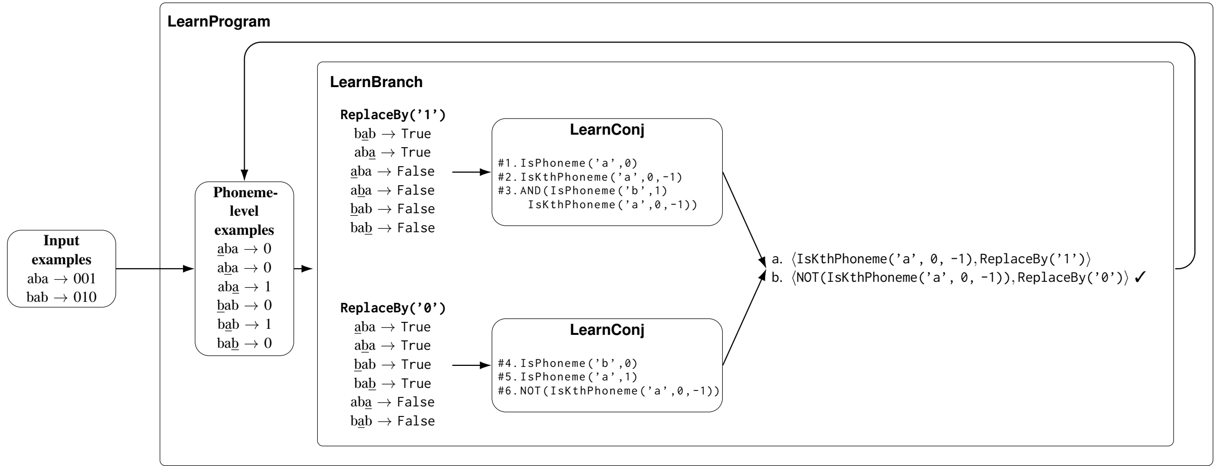

Figure 3: Illustration of the synthesis algorithm on a hypothetical case where the stress is on the last vowel, using the BASIC DSL. The input examples are first used to generate phoneme-level examples. The LearnProgram procedure then learns a decision list for the phoneme-level examples through calls to LearnBranch. The LearnBranch procedure iterates through different candidate transformations (such as ReplaceBy('0') and ReplaceBy('1')). For each transformation, the LearnConj procedure produces candidate conjunctions for when the transformation applies and when it does not. The candidate which is true for the most number of cases where the transformation applies, and none of the cases where it does not, is chosen. Here, this is #1 for the ReplaceBy('1') action and #6 for the ReplaceBy('0') action. The predicate-action pair which solves the most examples (here b) is then added to the decision list, and the LearnBranch procedure is called again on the unsolved examples.

for examples of words with stress patterns marked. All the words collected from these sources have the stress marking attested in the source – there are no cases of a given rule being used to predict the stress pattern on words.

Once words and the corresponding stress patterns are collected for a language, the set of words is split into two parts – one to be used for synthesizing programs (the *training* split) and the other (the *test* split) to be used for evaluating the synthesized rules. We ensure that all test examples are marked with a stress rule that is attested in the training examples.

We also use data from the stress problems presented in Vaduguru et al. (2021). These problems are chosen from the Linguistics Olympiads, a set of contests in linguistics for high school students. They present a task that is of a similar form to the tasks we study in this paper. These problems present words from a language with stress marked, and require a solver to use this data to infer the stress rules for that language, just as we do for the program synthesis system in this work. For these problems, we preserve the train-test splits from Vaduguru et al. (2021).

In total, we have data from 34 languages – 28 from the data we collect, and 6 from Linguistics Olympiad problems. Each language has between 5 and 33 training examples, with an average of 11.3, and between 2 and 16 test examples, with an average of 4.8.

4 Experiments

As described in Section 3, each language has a number of pairs – of word and stress pattern – in the training split. These are provided to the synthesis system, which produces a program. Given that the system checks shorter programs before longer ones, programs that are found after a long search are likely to be overfit to the given examples, and unlikely to generalize to unseen cases. This is why we terminate the synthesis if a program isn't found within 60 minutes. We also observe that for most of the languages, synthesis terminates well before this limit. For each language, we experiment with each of the 4 DSLs described in Section 2.2.

We also experiment with two neural sequence-to-sequence baselines, based on the LSTM and the Transformer architecture respectively. We use the implementation made available by Wu (2020), and train models with the same hyperparameters as in Vaduguru et al. (2021).

To evaluate the synthesized programs, we consider the output of the program on words in the test split. We only consider cases where the predicted stress pattern exactly matches the ground

truth stress pattern as correct, and compute the fraction of samples for which the predictions are correct – the *accuracy* of the program on the test set. We report the average accuracy for the set of languages we consider. We also report the average accuracy separately for data from each source – data which we collect and that chosen from Linguistics Olympiads – to observe any differences based on the source of data.

Additionally, we report the number of languages for which a synthesizer acheives a test accuracy of 100% or over 50%. This allows us to count the number of languages for which the synthesizers infer all, or a substantial fraction of, the rules of stress placement.

4.1 Results

The results obtained are shown in Tables 2 and 3. Language-wise results are presented in Appendix A. As expected, we see that neural baselines achieve low scores (except LSTM models on data from the Olympiads). Using program synthesis allows for significant gains over these baselines.

We observe that providing no information beyond the identity of the phonemes is not sufficient to infer correct rules. This is seen in the low overall accuracy obtained using the BASIC DSL, and the fact that it doesn't achieve perfect test accuracy for any of the languages.

Providing the DSL with just the distinction between consonants and vowels results in a big jump in performance. The CV DSL achieves a much higher average test accuracy, and is able to infer the rules fully in a number of languages.

Since stress placement is determined based on syllables, it is not surprising that encoding distinctions relevant to syllable weight hierarchies, such as vowel length and open/closed-ness of syllables, achieves the best performance. The SYLLABLE DSL achieves the highest average test accuracy, and the infers the rules fully in the highest number of languages.

While providing access to other features of phonemes in the FEATURE DSL does improve upon providing only the consonant-vowel distinctions, we see that it does not help as much as providing access to syllable-level distinctions.

We also note the difference between different sources here. Since Linguistics Olympiad problems are intended as reasoning challenges where solvers have to infer rules, they pose a more difficult learning challenge for our program synthesis system. This is seen in the lower accuracy obtained using all the DSLs for these languages. We also note that in these problems, access to syllable-level distinctions provides a larger gains relative to access to only consonant-vowel distinctions.

5 Analysis

We examine the synthesized programs for specific languages to understand the reasons for different levels of performance when using different DSLs, and illustrate patterns in failures due to properties of the DSL.

5.1 Benefits of the consonant-vowel distinction

We see that providing the synthesizer access to the distinction between vowels and consonants can improve its performance significantly. A synthesizer that does not have access to these needs to infer from the data alone that different vowels may behave in the same way, and that the behaviour may be common in a variety of contexts. In the absence of a large amount of data to provide negative evidence that occurrence in a specific context determines the application of a rule, the synthesizer tends to discover incorrect rules.

Consider the example of Lezgian. Stress is Lezgian is always placed on the second syllable of a word. Using the BASIC DSL, the system discovers rules such as

```
IfThenElse(
 And(PrefixContainsPhoneme('a', v, i),
 And(PrefixContainsPhoneme('l', v, i),
     Not(
     IsKthPhoneme('f', 0, 0, v, i)
     ))),
    ReplaceBy('1'))
```

This rule places stress on a phoneme if the prefix of the word up to the phoneme contain /a/ and /l/, and it is not the first occurrence of /f/ in the word. It also learns the rule

```
IfThenElse(
    SuffixContainsPhoneme('i', v, i),
    ReplaceBy('0'))
```

which does not place stress on a phoneme if the phoneme /i/ occurs after it in the word. This would lead to incorrect predictions if /i/ occurs in the third syllable of the word. Such rules are clearly overfit to the training data, and do not generalize well.

Languages	BASIC	CV	SYLLABLE	FEATURE	LSTM	Transformer
All	18.9	46.4	60.8	52.8	15.0	12.7
– Ours	18.8	52.8	63.9	57.1	13.2	12.8
– Olympiad	19.4	16.7	46.1	32.2	23.2	12.1

Table 2: Average accuracy across languages for each of the different DSLs for the entire set of languages and grouped by source of data.

Languages	BASIC		CV		SYLLABLE		FEATURE	
	$= 100\%$	$\geq 50\%$	$= 100\%$	$\geq 50\%$	$= 100\%$	$\geq 50\%$	$= 100\%$	$\geq 50\%$
All	0	7	8	17	12	21	11	18
– Ours	0	6	7	16	11	18	10	17
– Olympiad	0	1	1	1	1	3	1	1

Table 3: Number of languages where the system obtains perfect test accuracy, or test accuracy over 50%.

On the other hand, with the CV DSL, just the rules

```
IfThenElse(
    IsKthVowel(0, 1, v, i),
    ReplaceBy('1'),
    ReplaceBy('0'))
```

are learnt, which place stress on a phoneme if it is the second vowel (indexing starts at 0), and does not in all other cases. This illustrates the importance of access to such phonological distinctions when rules need to be learnt from a small amount of data.

5.2 Benefits from syllable-level distinctions

The benefits of being able to refer to syllable-level information in rules is visible in the programs synthesized for Sio. Stress in Sio depends on the weight of the syllable. If the final syllable of the word is a heavy syllable, it is stressed. If not heavy, the penultimate syllable is stressed. One of the rules the SYLLABLE grammar learns is

```
IfThenElse(
    Not(SuffixContainsDiphthong(v, i)),
    ReplaceBy('1'))
```

While there are other constraints to placement, this rule works towards ensuring that if the final syllable contains a diphthong (which is part of a heavy syllable), it is not stressed incorrectly.

To infer a rule about diphthongs correctly within the CV DSL, predicates about the first vowel have to be taken in conjunction with predicates about the second vowel, and this conjunction has to be distinguished from many other competing conjunctions which may also be consistent with the data. If other conjunctions which don't generalize beyond the training data are simpler, these are ranked higher and incorrectly chosen. Allowing the DSL to distinguish concepts such as diphthongs thus allows for learning simpler rules in such situations.

5.3 Incorrect generalizations

However, providing access to syllable-level distinctions may also encourage the synthesizer to discover incorrect generalizations. We see this in the case of Tzutujil. Stress in Tzutujil is placed on the final syllable of a word. With the CV DSL, the following rules are learnt.

```
IfThenElse(
  And(SuffixContainsVowel(v, i),
  And(IsKthVowel(0, -2, v, i),
      IsKthVowel(1, -1, v, i))),
    ReplaceBy('1'))

IfThenElse(
  And(Not(SuffixContainsVowel(v, i)),
      IsKthConsonant(-1, 0, v, i)),
    ReplaceBy('1'))

IfThenElse(
  And(Not(SuffixContainsVowel(v, i)),
      IsKthVowel(0, 1, v, i)),
    ReplaceBy('1'))
```

The first rule checks that if the suffix of the word after a phoneme to be stressed contains a vowel, it is the last vowel in the word. This is a case where the rule for a diphthong is discovered in the CV DSL. The other two rules ensure that a non-final vowel is not stressed by checking that the suffix doesn't contain any vowels.

The SYLLABLE DSL on the other hand discovers the rule

```
IfThenElse(
    Not(IsOpenSyllableVowel(0, v, i)),
    ReplaceBy('1'))
```

which incorrectly places stress on any vowel that is part of an open syllable. This results in the SYLLABLE DSL performing worse than the CV DSL for Tzutujil.

5.4 Insufficient constraints for stress placement

A common reason for failure is the failure to learn sufficient constraints for the application of rules. This results in sets of rules which allow primary stress to be placed on multiple phonemes, or on no phonemes, both of which are incorrect. We see examples of this in the program synthesized for stress in Cofan, where the penultimate syllable of the word is stressed.

Using the CV DSL, the following are some of the rules that are synthesized.

```
IfThenElse(
 And(IsKthVowel(0, 1, v, i),
     PrefixContainsPhoneme('k', v, i)),
 ReplaceBy('1'))

IfThenElse(
 And(PrefixContainsPhoneme('s', v, i),
     IsKthVowel(0, 0, v, i)),
 ReplaceBy('1')
```

Neither of these rules are sufficiently general, and rely on the presence of /k/ or /s/ in the prefix of the word up to the phoneme, neither of which is not relevant to the placement of stress. However, another problem is that there is no constraint that prevent both these rules applying to the same word. This occurs for the Cofan word /sokɨ/. The program incorrectly predicts that both syllables in this word receive primary stress, which is not allowed.

The program synthesized with the SYLLABLE DSL for the same data includes the rule

```
IfThenElse(
    IsKthConsonant(-1, -2, v, i),
 ReplaceBy('1'))
```

This rule places stress on the phoneme following the penultimate consonant of the word. When the word ends with two open syllables, this rule correctly predicts stress. However, for a word such as /ʔaiʔpa/, this rule does not apply. When other rules also fail to apply, as is the case for this word, no phoneme is predicted to be stressed. This violates the requirement that at least one syllable receive primary stress.

6 Related work

6.1 Program synthesis for linguistics

Barke et al. (2019) and Ellis et al. (2015) present program synthesis as a method for learning morphophonological rules from examples. They assume the existence of underlying forms, and infer rules of inflection that map an underlying form to the surface form using program synthesis. These rules operate directly on features of the phoneme, and not on the surface form of the words.

Sarthi et al. (2021) apply program synthesis to the problem of grapheme-to-phoneme conversion in Hindi and Tamil, which they pose as a string-to-string transformation task. Their design of the domain-specific languages captures specific phonological processes in Hindi and Tamil.

Vaduguru et al. (2021) apply program synthesis to learning phonological rules for string-to-string transformations from a small number of examples. They show that the method can be used to learn rules for various phonological phenomena like morphophonological rules, phonological rules relating similar languages, and stress rules in a challenging set of problems drawn from the Linguistics Olympiads. In this work, we focus on learning rules for stress placement alone, which allows us to specialize the DSL and investigate the effect of encoding phonological knowledge in the DSL.

6.2 Learning rules of phonological stress

Dresher and Kaye (1990) develop a system that learns stress patterns within the principles and parameters framework. Given words and the structure of syllables in these words, their method learns the parameters for principles relevant to the placement of stress.

Gupta and Touretzky (1992) propose a perceptron-based method for learning stress rules for empirical data. They propose a model that takes as input the weight of a syllable and predicts a value corresponding to the type of stress on the syllable.

Heinz (2006) proposes a method to learn rules for quality-insensistive stress, where the stress pattern depends only on the position and not on the weight of a syllable. Using the property of neighbourhood-distinctness, they propose a method

that learns a finite-state machine to model stress patterns.

While these works consider the learnability of stress patterns using a model that assumes certain features or properties of the input to be available, we propose a generic method where the availability of features can be controlled, and learning of abstract composite concepts like syllable weights from various primitive concepts can be investigated.

7 Conclusion

In this paper, we explore the problem of learning rules for the placement of phonological stress from only a few examples using program synthesis. We pose the problem as one of learning rules in the form of programs for string-to-string transformations. By designing the domain-specific language in which the rules are synthesized, we can control the amount of linguistic information available to the synthesizer.

We use the allowance to explicity provide the learning algorithm access to linguistic information to investigate how different linguistic concepts influence the rules that are learnt from data. To do this, we develop a generic program synthesis algorithm, and different domain-specific languages in which programs are synthesized. Each algorithm provides access to a different set of phonological classes, which can be used to identify phonemes that share common features.

We find that given a small number of examples, a synthesizer that doesn't have access to linguistic information beyond phoneme identity is unable to learn any useful rules. However, distinguishing consonants and vowels proves extremely useful, and distinguishing different types of syllables proves even more so.

Thus, using synthesis of rules for stress as a case study, we show how program synthesis can be used as a way to compare how different primitive concepts can be combined to learn rules for the same data using the same learning algorithm. Such methods can therefore be used to analyze what concepts are necessary to learn various rules from a limited number of samples, without changing the way in which these concepts are combined. Since program synthesis results in human-readable programs, we can also understand how primitive concepts are combined based on the data.

References

Lucas F.E. Ashby, Travis M. Bartley, Simon Clematide, Luca Del Signore, Cameron Gibson, Kyle Gorman, Yeonju Lee-Sikka, Peter Makarov, Aidan Malanoski, Sean Miller, Omar Ortiz, Reuben Raff, Arundhati Sengupta, Bora Seo, Yulia Spektor, and Winnie Yan. 2021. Results of the second SIGMORPHON shared task on multilingual grapheme-to-phoneme conversion. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 115–125, Online. Association for Computational Linguistics.

Shraddha Barke, Rose Kunkel, Nadia Polikarpova, Eric Meinhardt, Eric Bakovic, and Leon Bergen. 2019. Constraint-based learning of phonological processes. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6176–6186, Hong Kong, China. Association for Computational Linguistics.

Noam Chomsky and Morris Halle. 1968. The sound pattern of english.

B.Elan Dresher and Jonathan D. Kaye. 1990. A computational learning model for metrical phonology. *Cognition*, 34(2):137–195.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2015. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Rob Goedemans, Jeffrey Heinz, and Harry Van der Hulst. 2014. Stresstyp2. *University of Connecticut, University of Delaware, Leiden University, and the US National Science Foundation*.

Matthew Gordon. 2002. A factorial typology of quantity-insensitive stress. *Natural Language & Linguistic Theory*, 20(3):491–552.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Prahlad Gupta and David Touretzky. 1992. A connectionist learning approach to analyzing linguistic stress. In *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann.

Jeffrey Heinz. 2006. Learning quantity insensitive stress systems via local inference. In *Proceedings of the Eighth Meeting of the ACL Special Interest Group on Computational Phonology and Morphology at HLT-NAACL 2006*, pages 21–30, New York City, USA. Association for Computational Linguistics.

Dileep Kini and Sumit Gulwani. 2015. Flashnormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 776–783. AAAI Press.

Tiago Pimentel, Maria Ryskina, Sabrina J. Mielke, Shijie Wu, Eleanor Chodroff, Brian Leonard, Garrett Nicolai, Yustinus Ghanggo Ate, Salam Khalifa, Nizar Habash, Charbel El-Khaissi, Omer Goldman, Michael Gasser, William Lane, Matt Coler, Arturo Oncevay, Jaime Rafael Montoya Samame, Gema Celeste Silva Villegas, Adam Ek, Jean-Philippe Bernardy, Andrey Shcherbakov, Aziyana Bayyr-ool, Karina Sheifer, Sofya Ganieva, Matvey Plugaryov, Elena Klyachko, Ali Salehi, Andrew Krizhanovsky, Natalia Krizhanovsky, Clara Vania, Sardana Ivanova, Aelita Salchak, Christopher Straughn, Zoey Liu, Jonathan North Washington, Duygu Ataman, Witold Kieraś, Marcin Woliński, Totok Suhardijanto, Niklas Stoehr, Zahroh Nuriah, Shyam Ratan, Francis M. Tyers, Edoardo M. Ponti, Grant Aiton, Richard J. Hatcher, Emily Prud'hommeaux, Ritesh Kumar, Mans Hulden, Botond Barta, Dorina Lakatos, Gábor Szolnok, Judit Ács, Mohit Raj, David Yarowsky, Ryan Cotterell, Ben Ambridge, and Ekaterina Vylomova. 2021. Sigmorphon 2021 shared task on morphological reinflection: Generalization across languages. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 229–259, Online. Association for Computational Linguistics.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. *SIGPLAN Not.*, 50(10):107–126.

Partho Sarthi, Monojit Choudhury, Arun Iyer, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram Rajamani. 2021. ProLinguist: Program Synthesis for Linguistics and NLP. *IJCAI Workshop on Neuro-Symbolic Natural Language Inference.*

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. thesis, University of California, Berkeley.

Saujas Vaduguru, Aalok Sathe, Monojit Choudhury, and Dipti Sharma. 2021. Sample-efficient linguistic generalizations through program synthesis: Experiments with phonology problems. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 60–71, Online. Association for Computational Linguistics.

Shijie Wu. 2020. Neural transducer. `https://github.com/shijie-wu/neural-transducer/`.

Shijie Wu, Edoardo Maria Ponti, and Ryan Cotterell. 2021. Differentiable generative phonology.