Program Synthesis for Linguistic Rules

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computational Linguistics
by Research

by

Saujas Srinivasa Vaduguru
20171098
saujas.vaduguru@research.iiit.ac.in

International Institute of Information Technology
Hyderabad – 500 032, INDIA
June 2022

Copyright © Saujas Srinivasa Vaduguru, 2022
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Program Synthesis for Linguistic Rules" by Saujas Srinivasa Vaduguru, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Dr. Monojit Choudhury	Adviser: Prof. Dipti Misra Sharma

To my parents

Acknowledgments

I am grateful to several people for their guidance and support towards and during my work on this thesis.

My adviser, Monojit Choudhury, has been an incredible mentor both in my research and beyond. He sparked my curiosity about computational linguistics when I was still in school and has helped develop it into a deep research interest during my undergraduate and masters years. It was his idea that sparked the project that eventually led to this thesis, and he graciously agreed to guide my work on it. As my thesis adviser, he led me through the research process, providing the necessary guidance and support and also being patient and receptive to my ideas. He has shaped my development as a researcher in profound ways, and for this, I'm immensely grateful.

I am also grateful to my other adviser, Dipti Misra Sharma, for supporting and guiding my work on this project. Without her encouragement, this project might never have gone as far as it did. In working on this, she has given me a lot of freedom to pursue ambitious goals and difficult problems, and this work wouldn't have been possible without that.

I'd also like to thank my collaborators on this project, Aalok Sathe and Partho Sarthi. They have provided invaluable help and contributed to the development of these ideas in significant ways.

I am also thankful to the Pāṇini Linguistics Olympiad community, and to the broader Linguistics Olympiad community as a whole. The Olympiad kindled my interest in linguistics when I was a participant and has been a central part of what I've worked on in my research. I've also learned so much by participating in the Olympiad, helping organise it, and mentoring contestants.

Beyond the work on this thesis, a number of people have influenced my development as a researcher, and I would be remiss not to acknowledge them here. Evan Pu is one of the most interesting researchers I know. His picking up on an innocuous question I asked during NeurIPS 2020 has led to some of the most exciting research I have worked on so far, and collaborating with him has been an amazing experience. The summer (and more) I spent at Sarath Chandar's lab was an incredible learning experience. He is a versatile researcher and an extremely kind mentor, and has been a generous and supportive guide in pursuing a variety of ideas. During this period, I also had the opportunity to spend time with and learn from Prasanna Parthasarathi, whose enthusiasm to explore a variety of deep questions about language and learning has shaped how I think about them.

My extensive discussions with Shyamgopal Karthik and Ameya Prabhu have taught me a lot about machine learning, and even more about research. As a young student, I found their enthusiasm for

research and their extensive knowledge inspiring. I am extremely grateful for all the support and advice they have given me in navigating college and research.

I have also enjoyed discussing NLP, linguistics, and research with friends, including Aditya Yadavalli, Alok Debnath, Shashwat Goel, Shelly Jain, and Ujwal Narayan, and I'm grateful for the chance to learn from them.

I'm also grateful to have had supportive friends in college – AadilMehdi Sanchawala, Ajay Shrihari, Anirudha Ramesh, Aryan Sakaria, Athreya Chandramouli, Jainam Khakhra, Mahathi Vempati, Mukund Choudhary, Rahul Sajnani, Rohan Chacko, Sayar Ghosh Roy, Souvik Banerjee, Sriven Reddy, Zubair Abid and more.

In my time spent away from home in Hyderabad, my uncle and grandmother provided a much-needed support system. I'm grateful to them for helping me feel at home even when I was away and taking good care of me.

It almost goes without saying that this work wouldn't be possible without the incredible efforts of my parents. They have been ardent and unrelenting supporters of my interest in research and have done everything they can to enable my success. They have celebrated the good times with me and helped me get through more difficult times. For all this and more, I am immensely thankful.

Abstract

Recent work in NLP has focused on applying powerful neural sequence models to various learning problems. These neural models excel at extracting statistical patterns from large amounts of data but struggle to learn patterns or reason about language from only a few examples. We ask the question: Can we learn explicit rules that generalize well from only a few examples?

We explore this question by viewing linguistic rules as *programs* that operate on linguistic forms. This allows us to tackle the problem of learning linguistic rules using *program synthesis* – a method to learn rules in the form of programs in a domain-specific language (DSL). We develop a synthesis model to learn phonological rules as programs in a DSL. In addition to being highly sample-efficient, our approach generates human-readable programs and allows control over the generalizability of the learned programs.

We test the ability of our models to generalise from only a few training examples using our new dataset of problems from the Linguistics Olympiad. These problems are tasks from contests for high school students around the world that require inferring linguistic patterns from a small number of given examples. These problems are a challenging set of tasks that require strong linguistic reasoning ability.

Having shown that program synthesis can be used to learn phonological rules in highly data-constrained settings, we use the problem of phonological stress placement as a case to study how the design of the domain-specific language influences the generalisation 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 learned from a small number of examples.

Contents

Chapter Page

1 Introduction . . . 1
  1.1 Thesis organisation . . . 2

2 Linguistic Problems . . . 4
  2.1 Linguistics Olympiads . . . 4
    2.1.1 Rosetta Stone-style problems . . . 5
  2.2 Meta-linguistic awareness . . . 5
  2.3 Phonology problems . . . 6
  2.4 Dataset . . . 6
    2.4.1 Dataset statistics . . . 7

3 Learning from Limited Data . . . 8
  3.1 Abstract reasoning tasks . . . 8
    3.1.1 Compositionality . . . 9
    3.1.2 Program synthesis approaches . . . 10
  3.2 Phonology and morphology from limited data . . . 10
    3.2.1 Program synthesis approaches . . . 11

4 Program Synthesis . . . 13
  4.1 Dimensions of program synthesis . . . 13
  4.2 FlashMeta . . . 15

5 Learning Phonological Rules as String Transformations . . . 18
  5.1 Program synthesis . . . 18
    5.1.1 Phonological rules as programs . . . 18
    5.1.2 Domain-specific language . . . 19
    5.1.3 Synthesis algorithm . . . 20
    5.1.4 Structure of the problems . . . 22
  5.2 Experiments . . . 23
    5.2.1 Baselines . . . 23
    5.2.2 Program synthesis experiments . . . 23
    5.2.3 Metrics . . . 24
    5.2.4 Results . . . 24
  5.3 Analysis . . . 24
    5.3.1 Features aid generalisation . . . 25

5.3.2 Correct programs are short . . . 26
5.3.3 Using features . . . 26
5.3.4 Multi-pass rules . . . 27
5.3.5 Selecting spans of the input . . . 27
5.3.6 Global constraints . . . 27
5.4 Conclusion . . . 28

6 Learning Rules for Phonological Stress Placement . . . 29
6.1 Program synthesis . . . 29
6.1.1 Stress rules as programs . . . 29
6.1.2 Domain-specific languages . . . 30
6.1.2.1 Classes of phonemes . . . 31
6.1.2.2 Predicate types . . . 31
6.1.3 Synthesis algorithm . . . 34
6.2 Dataset . . . 35
6.3 Experiments . . . 35
6.3.1 Results . . . 36
6.4 Analysis . . . 37
6.4.1 Benefits of the consonant-vowel distinction . . . 37
6.4.2 Benefits from syllable-level distinctions . . . 38
6.4.3 Incorrect generalisations . . . 38
6.4.4 Insufficient constraints for stress placement . . . 39
6.5 Related work . . . 40
6.6 Conclusion . . . 41

7 Conclusions . . . 42

Appendix A: . . . 44
A.1 Olympiad data . . . 44
A.2 Examples . . . 44

Appendix B: . . . 48
B.1 Neural . . . 48
B.2 WFST . . . 48

List of Figures

Figure Page

4.1 An illustration of the search performed by the FlashMeta algorithm. The blue boxes show the specification that an operator must satisfy in terms of input-output examples, with the input token underlined in the context of the word. The `Inverse Semantics` of an operator is a function that is used to infer the specification for each argument of the operator based on the semantics of the operator. This may be a single specification (as for predicate) or a disjunction of specifications (as for token and offset). The algorithm then recursively searches for programs to satisfy the specification for each argument, and combines the results of the search to obtain a program. The search for the transformation – rule – in an `IfThen` statement proceeds similarly to the search for a predicate. Examples of programs that are inferred from a specification are indicated with $\Longrightarrow$. A dashed line between inferred specifications indicates that the specifications are inferred jointly. 17

5.1 `IfThen-Else` statements in the DSL 19
5.2 An illustration of the synthesis algorithm. FM is FlashMeta, which synthesises rules which are combined into a disjunction of rules by NDSyn. Here, rule `#1` is chosen over `#4` since it uses the more general concept of the `voice` feature as opposed to a specific token and thus has a higher ranking score. 22
5.3 Number of rules plotted against EXACT score 25

6.1 `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. 31
6.2 Classes available to each DSL – BASIC, CV, SYLLABLE, and FEATURE. 32
6.3 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. 33

List of Tables

Table Page

1.1 Verb forms in Mandar (McCoy, 2018) 1

2.1 A few examples from different types of Linguistics Olympiad problems. '?' represents a cell in the table that is part of the test set. 6

5.1 Predicates that are used for synthesis. The predicates are applied to a token x that is at an offset i from the current token in the word w. The offset may be positive to refer to tokens after the current token, zero to refer to the current token, or negative to refer to tokens before the current token. 20
5.2 Transformations that are used for synthesis. The transformations are applied to a token x in the word w. The offset i for the Copy transformations may be positive to refer to tokens after the current token, zero to refer to the current token, or negative to refer to tokens before the current token. 21
5.3 Metrics for all problems, and for problems of each type. The CHFF score for stress problems is not calculated, and not used to determine the overall CHRF score. 23
5.4 Number of problems where the model achieves different thresholds of the EXACT score. 25

6.1 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. 30
6.2 Average accuracy across languages for each of the different DSLs for the entire set of languages and grouped by source of data. 36
6.3 Number of languages where the system obtains perfect test accuracy, or test accuracy over 50%. 36

A.1 Languages for which we collect data. 45
A.2 Olympiad problems. Kabardian is an IOL problem in which stress is presented as a side phenomenon, and is easier to deduce than in other IOL problems. 45
A.3 Creek stress 46
A.4 Kabardian stress 47

Chapter 1

Introduction

In the last few years, the application of deep neural models has allowed rapid progress in NLP. Tasks in phonology and morphology have been no exception to this, with neural encoder-decoder models achieving strong results in recent shared tasks in phonology (Gorman et al., 2020) and morphology (Vylomova et al., 2020). They have 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 structures and patterns these models learn from data is a non-trivial task. In this paper, we explore the problem of learning interpretable phonological and morphological rules from only a small number of examples, a task that humans are able to perform.

to *V*	**to be *V*ed**
mappasuŋ	dipasuŋ
mattunu	ditunu
?	ditimbe
?	dipande

Table 1.1: Verb forms in Mandar (McCoy, 2018)

Consider the example of verb forms in the language Mandar presented in Table 1.1. How would a neural model tasked with filling the two blank cells do? The data comes from a language that is not represented in large-scale text datasets that could allow the model to harness pretraining, and the number of samples presented here is likely not sufficient for the neural model to learn the task.

However, a human would fare much better at this task even if they didn't know Mandar. Identifying rules and patterns in a different language is a principal concern of a descriptive linguist (Brown and Ogilvie, 2010). Even people who aren't trained in linguistics would be able to solve such a task, as evidenced by contestants in the Linguistics Olympiads[1], and general-audience puzzle books (Bellos, 2020). In addition to being able to solve the task, humans would be able to express their solution explicitly in terms of rules, that is to say, a *program* that maps inputs to outputs.

[1] https://www.ioling.org/

In this thesis, we aim to devise methods that can learn linguistic rules from a small number of examples, and study the effect of prior knowledge on how the learned rules generalise to unseen examples. We explore a different approach to learning linguistic patterns from data – by viewing rules as programs that operate on linguistic forms. In this view, we can cast the task of learning linguistic patterns as *program synthesis*. Program synthesis (Gulwani et al., 2017) is a method that can be used to learn programs that map an input to an output in a *domain-specific language* (DSL). It has been shown to be a highly sample-efficient technique to learn interpretable rules by specifying the assumptions of the task in the DSL (Gulwani, 2011).

The contributions of this thesis are as follows:

- Applying program synthesis methods to learning phonological rules as string transformation programs given only a few examples
- Curating a set of phonology problems from Linguistic Olympiads suitable to evaluate a system that is tasked with learning linguistic patterns from small amounts of data
- Showing the efficacy of this method in learning rules to solve challenging problems from Linguistics Olympiads
- Exploring how the design of the domain-specific language and the linguistic knowledge encoded into the synthesis procedure in synthesising rules for stress placement

1.1 Thesis organisation

In Chapters 2 to 4, we lay out the relevant background on Linguistics Olympiads and program synthesis, and discuss related work on learning from limited amounts of data.

Chapter 2 We describe Linguistics Olympiads and discuss the challenges involved in solving problems from these Olympiads. We discuss the types of problems we tackle in this work and describe the dataset of problems we compile to evaluate our system.

Chapter 3 We review work on learning abstract rules from limited data. We discuss two broad views of the task of solving problems like those that appear in the Linguistics Olympiads – as a task of learning rules for abstract reasoning tasks, and as learning linguistics patterns (specifically phonological and morphological) patterns from a limited amount of data.

Chapter 4 We present a short primer on program synthesis. We discuss different dimensions of the program synthesis task and discuss the choice of program synthesis approach for the learning problem we tackle. We also describe a key component of the program synthesis approach used in later chapters.

In Chapters 5 and 6 we present the synthesis formalism, describe the algorithm, discuss the experimental methods, and analyse the results.

Chapter 5 We 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. We demonstrate a method for learning different types of rules, including morphophonology, transliteration, and phonological stress.

Chapter 6 We extend the formulation of phonological stress placement as a string-to-string transformation problem from Chapter 5 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 generalisation from some given data.

Chapter 2

Linguistic Problems

In this chapter, we introduce and discuss Linguistics Olympiads and describe the data collected for our experiments.

2.1 Linguistics Olympiads

Linguistic problems are a genre of composition that presents linguistic facts and phenomena in enigmatic form (Derzhanski and Payne, 2010). The most common use of these problems is in contests for secondary school students such as the North American Computational Linguistics Olympiad (in the USA and Canada), the Panini Linguistics Olympiad (in India), and the International Linguistics Olympiad (IOL). These problems satisfy two conditions (Bozhanov and Derzhanski, 2013):

- They do not require knowledge (including knowledge of linguistic theory) beyond what is taught in the secondary school curriculum
- They must have a unique, unambiguous solution

Though these problems are designed such that they do not require specific linguistic knowledge, they require the ability to perform complex reasoning and spot general patterns using very few examples. The problems are also designed in a manner that ensures the patterns can be deduced from the data. This guarantee makes these problems a suitable challenge for algorithms that can learn to generalise from only a few examples.

In these problems, the solver is presented with examples of linguistic forms (words, phrases, sentences) and other forms derived by applying the rules to these forms. These forms, which we will refer to as the *data*, typically consist of 20-50 forms, and the problems are designed to provide the minimal number of examples required to infer the correct rules is presented (Şahin et al., 2020).

The *assignments* provide other linguistic forms, and the solver is tasked with applying the rules inferred from the data to these forms. The forms in the assignments are carefully selected by the designer to test whether the solver has correctly inferred the rules, including making generalisations to unseen

data. This allows us to see how much of the intended solution has been learnt by the solver by examining responses to the assignments.

The small number of training examples (data) tests the generalisation ability and sample efficiency of the system and presents a challenging learning problem for the system. The careful selection of test examples (assignment) lets us use them to measure how well the model learns these generalisations.

2.1.1 Rosetta Stone-style problems

One class of linguistic problems – dubbed Rosetta Stone problems – is particularly well suited for this kind of challenge. These problems consist of ordered matching expressions of two languages or language-like symbolic systems, so chosen as to enable deducing the regularities behind the correspondences Bozhanov and Derzhanski (2013). Often, these problems take the form of translation tasks, where phrases or sentences from two languages are given, and the solver has to infer the rules required to translate from one language to the other.

Şahin et al. (2020) present a set of Rosetta Stone-style problems from various Linguistics Olympiads based on translation tasks as a benchmark for how well modern systems perform on translation tasks given a very small number of examples. These problems require rules at various levels of linguistic processing – phonology, morphology, syntax, and semantics. A further challenge in these problems is the interaction between these different levels of processing, making these problems difficult benchmarks for learning linguistic regularities from data.

Unsurprisingly, they find that while humans are able to solve these tasks – as evidenced by the contestants in the Olympiads in which these problems appeared – even powerful NLP models perform quite poorly, getting hardly any translations correct.

In addition to the benchmark, and experimental results, Şahin et al. (2020) identify *meta-linguistic awareness* as a requirement to solve such problems, where the amount of data given is extremely small.

2.2 Meta-linguistic awareness

Şahin et al. (2020) build on Chomsky's (1975) definition of meta-linguistics as "the knowledge of the characteristics and structures of language". They argue that this meta-linguistic awareness – knowledge of how languages work in general, beyond any specific language – can help with learning new languages efficiently. In the context of NLP models, having greater meta-linguistic awareness can help with cross-lingual generalisation, and transferring knowledge between languages more easily.

The program synthesis approach we take allows us to encode certain forms of prior knowledge into the space in which we search for solutions. These can be viewed as building certain specific forms of meta-linguistic awareness into our models. This setting allows us to explicitly define what we consider to be meta-linguistic awareness, and understand what forms of this awareness are helpful in inferring rules

base form	negative form
joy	kas joyaːya’
biːlaw	kas bika’law
tipoysuːda	?
?	kas wurulaːla’

(a) Movima negation

Turkish	Tatar
bandır	mandır
yelken	cilkän
?	osta
bilezik	?

(b) Turkish and Tatar

Listuguj	Pronunciation
g’p’ta’q	gəbədaːx
epsaqtejg	epsaxteck
emtoqwatg	?
?	əmteskəm

(c) Micmac orthography

Aleut	Stress
tatul	`01000`
nətɣəlqin	`000010000`
sawat	?
qalpuqal	`00001000`

(d) Aleut stress

Table 2.1: A few examples from different types of Linguistics Olympiad problems. ‘?’ represents a cell in the table that is part of the test set.

from only a small number of samples. In the context of learning from data, meta-linguistic awareness can also be viewed as a form of *inductive bias*.

2.3 Phonology problems

Linguistics Olympiad problems, in general, are based on various types of linguistic phenomena – including phonology, morphology, syntax, and semantics – and often involve multiple linguistic phenomena that interact across levels of linguistic analysis.

In this thesis, we restrict our study to problems based on phonology. More specifically, we choose problems whose solution depends on rules that can be inferred based only on the *form* of the data, without relying on aspects of *meaning*. Olympiad problems often involve phenomena that require reasoning about semantic concepts and categories, such as tense, number, grammatical gender, etc. When solving these problems, humans rely on their knowledge of these concepts in the language(s) they already know. However, in designing a program synthesis system, especially one where we don’t assume access to extensive data for learning, defining such categories in a way that can be flexibly used across languages is a difficult, if not insurmountable, challenge. In choosing problems based on phonology, we restrict our investigation to a space where concepts can be defined based on classes of symbols, or in terms of a small set of well-understood features.

2.4 Dataset

We select 34 problems from various Linguistics Olympiads to create our dataset. We include publicly available problems that have appeared in Olympiads before. These problems are based on phonology,

and some aspects of the morphology of languages, as well as the orthographic properties of languages. We choose problems that only involve rules based on the symbols in the data, and not based on knowledge of notions such as gender, tense, case, or semantic role. These problems are based on the phonology of a particular language and include aspects of morphology and orthography, and maybe also the phonology of a different language. In some cases where a single Olympiad problem involves multiple components that can be solved independently of each other, we include them as separate problems in our dataset. As described in Section 2.3, these problems are chosen such that the underlying rules depend only on the given word forms, and not on inherent properties of the word like grammatical gender or animacy. The problems involve (1) inferring phonological rules in morphological inflection (Table 2.1a) (2) inferring phonological changes between multiple related languages (Table 2.1b) (3) converting between the orthographic form of a language and the corresponding phonological form (Table 2.1c) (4) marking the phonological stress on a given word (Table 2.1d). We refer to each of these categories of problems as morphophonology, multilingual, transliteration, and stress respectively. The dataset is available at `https://github.com/saujasv/phonological-generalizations`.

2.4.1 Dataset statistics

The dataset we present is highly multilingual. The 34 problems contain samples from 38 languages, drawn from across 19 language families. There are 15 morphophonology problems, 7 multilingual problems, 6 stress, and 6 transliteration problems. The set contains 1452 training words with an average of 43 words per problem and 319 test words with an average of 9 per problem. Each problem has a matrix that has between 7 and 43 rows, with an average of 23. The number of columns ranges from 2 to 6, with most problems having 2.

Chapter 3

Learning from Limited Data

In this chapter, we review different directions in learning from limited data. First, we consider work in the cognitive science and artificial intelligence communities on inducing abstract rules from limited data. Then, we review work from the natural language processing and computational linguistics communities on learning phonological and morphological rules from limited data.

3.1 Abstract reasoning tasks

Solving the types of problems we consider in this thesis can be viewed as an instance of the task of inferring abstract rules from limited data, with the data being in the form of strings of phonemes in a particular language. There has been significant work studying how humans infer abstract concepts and solve such reasoning challenges, and also in developing computational models of such reasoning processes. Moreton et al. (2017) also cast phonological concept induction as an instance of general concept induction methods, connecting the elements of phonological rule learning and reasoning about abstract concepts that we examine in this work.

Chollet (2019) presents one such suite of abstract reasoning tasks called the *ARC challenge*. These tasks are intended to measure progress towards general, fluid, human-like intelligence. The tasks consist of sets of input-output pairs, each of which is a pair of grids. The cells of the grid may be coloured in different ways, forming a pattern. Each input-output pair demonstrates the application of a specific transformation to the input grid that produces the output grid. An agent that solves these tasks must infer the transformation, and successfully apply it to an input grid (for which the corresponding output is not provided) to obtain the correct output.

Johnson et al. (2021) cast the ARC challenge as an instance of human program induction, and measure how well humans perform on these abstract reasoning tasks.[1] They note that the ARC challenge incorporates “elements such as flexible hypothesis generation, few-shot learning, compositionality and program induction”, all of which are important aspects of solving the Olympiad problems we consider

[1] They find that humans are able to solve 80% of these tasks on average, which is far higher than the best performing model at the time of their writing, which solved 21% of the tasks.

as well. Another aspect of Olympiad problems that is also a feature of the ARC challenge is the presence of a generation component – the solver has to generate answers to the assignments from scratch. One theme that has been extensively explored in machine learning and artificial intelligence work that has sought to build models that can generalise from a small amount of data is compositionality.

3.1.1 Compositionality

Compositionality is the notion that a small number of primitives can be combined to create complex, composite structures. Lake et al. (2017) argue that the productivity afforded by compositionality is core to how humans can learn new concepts from a large space of possibilities. Compositionality is also an essential component of abstract rule-like structures, which humans can learn efficiently and use to generalise from only a few examples or demonstrations (Lake et al., 2019).

The ability of neural network (or *connectionist*) models to generalise compositionally in a systematic manner is yet to be established (Lake and Baroni, 2018). To tackle the issue of compositionality, a number of approaches have been proposed. Here, we review two classes of approaches – *meta-learning* and *data augmentation* – which have been applied to sequence-to-sequence problems in a few-shot learning setting.

Meta-learning is the notion of learning how to acquire a new task from only a few examples through training across many such tasks. Lake (2019) presents an approach using meta-learning to train a memory-augmented sequence-to-sequence model and introduce compositional inductive biases. Through training across many tasks, the model learns to use its memory to generalise systematically from only a few given examples of sequence-to-sequence transformations at test time.

Another approach to achieving compositional generalisation has been through data augmentation. Data augmentation refers to "strategies for increasing the diversity of training examples without explicitly collecting new data" (Feng et al., 2021). Andreas (2020) presents a data augmentation method designed to improve compositional generalisation by identifying phrases in the training data and substituting them in new environments. This simple approach improves the performance of black-box neural network models in a variety of tasks, including semantic parsing and low-resource language modeling. Akyürek et al. (2021) build upon the idea of data augmentation for compositional generalisation and propose using learned augmentations. They use a learned model to combine examples from training data – either by copying fragments or learning to insert appropriate examples in context – and resample to obtain an augmented training set. This approach allows for highly data-efficient generalisation. They study the setting of morphological analysis, similar to the task we consider, and find that their model can generalise to a new paradigm given as few as 8 examples of that paradigm.

Though meta-learning is a promising approach to learn from a small number of examples in a new task, it does require a large number of tasks to learn from. Data augmentation approaches may allow for learning new skills (such as inflecting in new morphological paradigms), or improve compositional

generalisation, but still requires a large amount of data. Given the constraints on the data we have, we adopt a program synthesis approach and design the DSL to enable desirable forms of generalisation.

3.1.2 Program synthesis approaches

Program synthesis approaches – including ones that incorporate learning, search, and some combination of the two – have shown significant promise in generalising from only a few examples.

Search-based program synthesis approaches have succeeded at learning rules from very few input-output examples. One of the most successful instances of this is the FlashFill system (Gulwani, 2011) that learns string transformations. When the ARC challenge was hosted on Kaggle, a popular machine learning contest platform, the best performing solution at the time used search-based program synthesis[2].

Neural models have also resulted in learning-based approaches to program synthesis. Devlin et al. (2017) presented a system that learnt to generate the string form of a program given a small set of input-output examples. Execution-guided approaches have also been successful at these string transformation tasks, using the result of executing partial programs to build larger, more complex programs (Ellis et al., 2019). Nye et al. (2020) present a method where a program synthesis-based approach is used to learn compositional rules for string transformation tasks, including mapping numeral expressions in language to the numeric value in a number of different languages.

3.2 Phonology and morphology from limited data

In this thesis, we focus on a subset of Linguistics Olympiad problems that deal specifically with phonology and morphology. In this section, we review work on learning linguistic rules in highly data-constrained settings.

Recent work on morphology has focused on the task of morphological (re)inflection, where the surface form, or stem, of a word and a target morphological paradigm are given, and the task is to learn to predict the surface form of the word inflected for the target paradigm.

Recent shared tasks on morphological reinflection, have seen a number of approaches that employ powerful neural sequence transduction models (Vylomova et al., 2020). Data augmentation-based approaches (Anastasopoulos and Neubig, 2019), in combination with ensembling and other training techniques have allowed for morphological reinflection models even for low-resource languages. Anastasopoulos and Neubig (2019) also explore the efficacy of transfer learning – training on a language with more available labelled data to improve performance on languages with little training data – for the reinflection task.

Kann and Schütze (2018) study the morphological reinflection task in the setting where even fewer labelled samples are available than in typical low-resource morphological reinflection settings, which

[2] https://github.com/top-quarks/ARC-solution

they term the *minimal-resource* setting. They consider this setting as an instance of paradigm completion, where given a partial paradigm of a lemma, the remaining forms are to be generated. They use transductive learning to use the partial paradigms at test time to improve predictions. This work is aimed at learning to generalise with a small number of paradigms, but the number of forms within a paradigm may not be minimal.

While the formulation of the morphological reinflection tasks is similar (when viewed as filling cells in a paradigm table), the full reinflection task may have instances where the answer cannot be determined only based on the given forms. For example, the Czech data from the SIGMORPHON 2017 shared task[3] includes the following training triples:

Source form	**Target form**	**Tags**
dvojka	*dvojky*	`N;GEN;SG`
svazek	*svazku*	`N;GEN;SG`

Here, the difference in the suffix depends on the gender of the base form. This information is not available in the paradigm tag.

Another focus of our investigation is the number of samples, where a small number of selected examples is provided to ensure the task is solvable. With Olympiad problems, we know that the test set forms can be deduced based on the other given (training set) forms. However, the reinflection task data has more samples, and selecting a subset of training samples to allow all test forms to be inferred is itself a challenging task.

Learning rules in extremely resource-constrained settings has also been explored in other computational linguistics tasks. Chaudhary et al. (2020) explore learning rules for morphosyntactic agreement. They use transfer learning-based approaches to dependency parsing to obtain dependency trees for sentences in a language where only a small amount of (labelled) data may be available. Based on the trees, they extract features that can be used to train a decision tree-based classifier for predicting agreement. Since decision trees are interpretable models the decision tree constitutes a set of rules for predicting agreement.

Finally, the use of a hand-crafted domain-specific language in program synthesis allows us to control the specific inductive biases that our models use to learn generalisable rules. Gildea and Jurafsky (1996) also study the problem of learning phonological rules from data and explicitly controlling generalisation behaviour through specifying inductive biases. However, we consider this problem in a setting where the number of samples available is far smaller.

3.2.1 Program synthesis approaches

Ellis et al. (2015) apply program synthesis to learning morphophonological rules in English. Given triples of lexeme, tense, and surface form (which constitute paradigm tables presented in a different

[3] `https://github.com/sigmorphon/conll2017/blob/master/all/task2/czech-train-low`

format), their model jointly infers regular rules that govern the morphophonological processes. The joint inference proves to be a computationally expensive solution, and this limitation is addressed by Barke et al. (2019). Their method breaks down the process into three steps – inferring the underlying form of the word, inferring changes between the underlying form and the surface form, and then inferring conditions in which these changes occur. This decomposition of the problem allows for much faster inference.

Ellis et al. (2015) and Barke et al. (2019) learn phonological rules to determine the surface forms of words in different paradigms by applying rules to an underlying form or stem. This is similar to the framework presented by Chomsky and Halle (1968), where the "phonological component is a system of rules [...] that relates surface structures [...] to phonetic representations."[4] Thus, to transform a word in one morphological paradigm to another, the underlying form of the word has to be inferred, and the rules that convert the underlying form to the surface form. However, these underlying forms are an abstraction posited as part of the learning process, and the strings corresponding to the underlying forms are also inferred alongside the rules. Thus, some prior work applying program synthesis to learn linguistic rules works within the theoretical framework described in Chomsky and Halle (1968).

In contrast, the approach we adopt in this thesis does not follow any particular theoretical framework. We design systems that learn rules to directly convert between surface forms. We do not infer underlying forms as an intermediate step in converting between these surface forms. We also assume access to features (which are part of the DSL used by Barke et al. (2019) and Ellis et al. (2015)) is optional, allowing us to observe the effect of access to features when learning general string transformation programs. This also allows us to test our approach on a more varied set of problems that involves aspects of morphology, transliteration, multilinguality, and stress by modeling them as instances of string transformations. Despite differing the theoretical motivations of the work, we adopt the formalism of conditional rewrite rules in this work (this formalism is described in greater detail in Chapter 5). Whereas prior work uses conditional rewrite rules to model the transformation from the string corresponding to the underlying form to surface form, we use them to directly model transformations between surface form strings.

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. In, Chapter 5 we adapt the method presented by Sarthi et al. (2021) to the task of learning phonological rules.

[4]Chomsky and Halle (1968) use *surface structure* to refer to the output of the syntactic process, what we refer to elsewhere as the underlying form for the phonological component, and *phonetic representation* is used to refer to the output of the phonological component, which we refer to elsewhere as the surface form.

Chapter 4

Program Synthesis

Program synthesis is "the task of automatically finding programs from the underlying programming language that satisfy (user) intent expressed in some form of constraints" (Gulwani et al., 2017). This definition highlights two important aspects of program synthesis – the underlying programming language, or *domain-specific language*, and the expression of intent in the form of constraints. Gulwani et al. (2017) identify three dimensions in program synthesis, which we discuss.

4.1 Dimensions of program synthesis

Intent There are many possible ways to specify intent in program synthesis. Given the specification, synthesis is the task of finding programs that are *consistent* with the specification. Some ways of specifying intent are:

- **Programs:** The intended program can be defined as one that behaves identically to another specified program. This form of specification finds application in tasks like superoptimisation – where the synthesiser is used to find a more efficient program that behaves identically to the intended program. (Bansal and Aiken, 2008).
- **Partial programs:** Another way of specifying intent is by providing partial programs that have *holes* that need to be filled by synthesised code. The SKETCH system (Solar-Lezama, 2008) is an example of a synthesiser that completes partial programs.
- **Natural language:** Intent can also be specified as a natural language utterance (Desai et al., 2016). Natural language may be used as the sole form of specification, or in conjunction with examples and other types of constraints in multimodal program synthesis (Ye et al., 2021).
- **Examples:** Examples are a popular form of specifying intent for program synthesis and the form we focus on in this thesis. Here, a small number of inputs, along with the result of executing the intended program on these inputs, is provided to the synthesiser. This has been used for building synthesisers that can find programs for tasks such as string transformations from just 2-3 examples

(Gulwani, 2011). We cast the task of solving Rosetta Stone-style Olympiad problems as finding the program that transforms forms in one language to another, or from one inflectional paradigm to another.

Search space The domain-specific language (DSL) specifies the space of programs that needs to be searched. By imposing some structure on the search space through the DSL, we can make the search procedure more efficient. Additionally, the design of the DSL allows us to encode the domain-specific knowledge into the space of programs. This ensures that the program space only contains programs that are feasible solutions, and eliminates many spurious solutions that may satisfy the given constraints. When working in challenging settings like learning rules from only a few input-output examples, the design of the DSL can improve synthesis significantly (we explore this in Chapter 6).

The ability to explicitly encode domain-specific assumptions gives program synthesis broad applicability to various tasks. In this thesis, we explore applying it to the task of learning phonological rules. Whereas previous work on rule-learning has focused on learning rules of a specific type (Brill, 1992; Johnson, 1984), the DSL in program synthesis allows learning rules of different types, and in different rule formalisms.

Search technique Given a form of specifying intent, and a space to search in, there are different ways of searching for the intended program.

- **Enumeration:** One way of synthesising programs is enumerating programs in some order (like in order of increasing size/complexity) until a program consistent with the specification is found. This is a simple technique for synthesis but can be difficult to scale.

- **Deduction:** Another way to search for programs is to break down the search problem into a collection of smaller search problems using rules of deduction, and combining the solutions to these. The FlashMeta algorithm is an instance of this class of methods, and we use this type of search for solving problems in later chapters. FlashMeta is described in greater detail in Section 4.2.

- **Constraint solving:** This way of synthesising programs involves generating logical constraints whose resolution results in a program consistent with the specification. This form of synthesis has been used in prior work by Barke et al. (2019) applying program synthesis to linguistic rules, which was discussed in Chapter 3.

- **Statistical methods:** Techniques involving statistical methods including machine learning (Menon et al., 2013), genetic programming (Weimer et al., 2009), MCMC sampling (Schkufza et al., 2013), and probabilistic inference (Jojic et al., 2006) have been studied in the context of program synthesis. Recent work has found that neural models, particularly neural sequence models, are powerful, robust synthesisers (Devlin et al., 2017; Nye et al., 2020).

An important concern in searching for programs is incorporating domain-specific preferences for some programs over others. Some preferences, such as a preference for shorter programs, are useful in a variety of domains. Incorporating domain-specific knowledge can result in more generalisable programs, and allow for learning from even a small number of examples. These preferences may be introduced in many ways. One common way is to use a ranking function, that imposes an ordering over the space of programs and breaks ties between multiple programs that may be consistent with a given specification. This ranking function may be learnt from data (Gulwani and Jain, 2017), or specified by hand (Gulwani, 2011). The FlashMeta algorithm allows specifying a ranking function by hand, and we use this since we do not have sufficient data for learning. Other ways of imposing preferences over programs include posing synthesis as a constrained *optimisation* problem rather than a constraint satisfaction problem. This approach was adopted by Barke et al. (2019).

4.2 FlashMeta

FlashMeta is a framework for inductive program synthesis. The framework allows for specifying a domain-specific language — defined by

- a *syntax* in the form of a context-free grammar, which has rules that expand non-terminal variables on the left hand side to the application of an operator to non-terminal variables on the right hand side
- a *semantics* which defines the semantics of each operator in the DSL, which the framework treats as black-box operators that can be defined by the user
- *witness functions* that define the *inverse semantics* of the operator, and allow for *deductive* search
- a *ranking function* that assigns a (sub-)program a real-valued score

This structure of FlashMeta allows it to leverage multiple forms of guidance in the search. The domain-specific language defines a syntax that can be used to guide search, as in SyGuS (**?**). The use of witness functions allows the task of synthesis to be broken down into smaller sub-tasks – a *deductive* process. The specification of a ranking function allows for imposing domain-specific preferences over consistent programs.

Now, we sketch the working of witness functions – a central part of FlashMeta's search procedure. The synthesis task is given by the DSL operator P and the specification of constraints $\mathcal{X}$ that the synthesised program must satisfy. The algorithm recursively decomposes the synthesis problem $(P, \mathcal{X})$ into smaller tasks $(P_i, \mathcal{X}_i)$ for each argument P_i to the operator. $\mathcal{X}_i$ is inferred using the inverse semantics of the operator P_i, which is encoded as a *witness function*. The inverse semantics provides the possible values for the arguments of an operator, given the output of the operator.

In our application, this specification is in the form of token-level examples. These examples are obtained from input-output pairs of strings by aligning the input and output at the token level. This

yields a set of pairs where the input is a 'central token' in context of other tokens, and the output is another token. The DSL operators are the predicates and transformations that are used in if-then rules. These rules use predicates to check whether a transformation can be applied to a token based on its identity and its context, and if it can, returns the result of applying the transformation to the token.

Consider a simplified setting where one predicate, `IsToken(w, c, o)`, can be used to check whether a token in the string `w` at an offset `o` from the central token is equal to the token `c`. In this setting, the token may be transformed by one of two operators – `ReplaceBy(x, c1, c2)` and `ReplaceAnyBy(x, c)`. `ReplaceBy` returns the token `c2` if `x` is equal to `c1`. `ReplaceAnyBy` always returns `c` as a replacement for `x`. Figure 4.1 illustrates the working of FlashMeta in this setting. The specific DSL structure used – including predicates, transformations, and the mechanism for converting between string-level examples and token-level examples – is described in detail in Chapters 5 and 6.

The FlashMeta framework also allows for defining a ranking function over programs and sub-programs by defining a score of an operator in terms of the scores for the arguments it takes (which may in turn be the result of applying another operator). In the example shown in Figure 4.1, the score for the `IsToken` operator is defined in terms of its arguments – a token, and an integer value for the offset. Similarly, the value for an `IfThen` construct is defined in terms of the score of the predicate and the score of the transformation that it takes as arguments. Details about the specific ranking function used are also presented in Chapters 5 and 6.

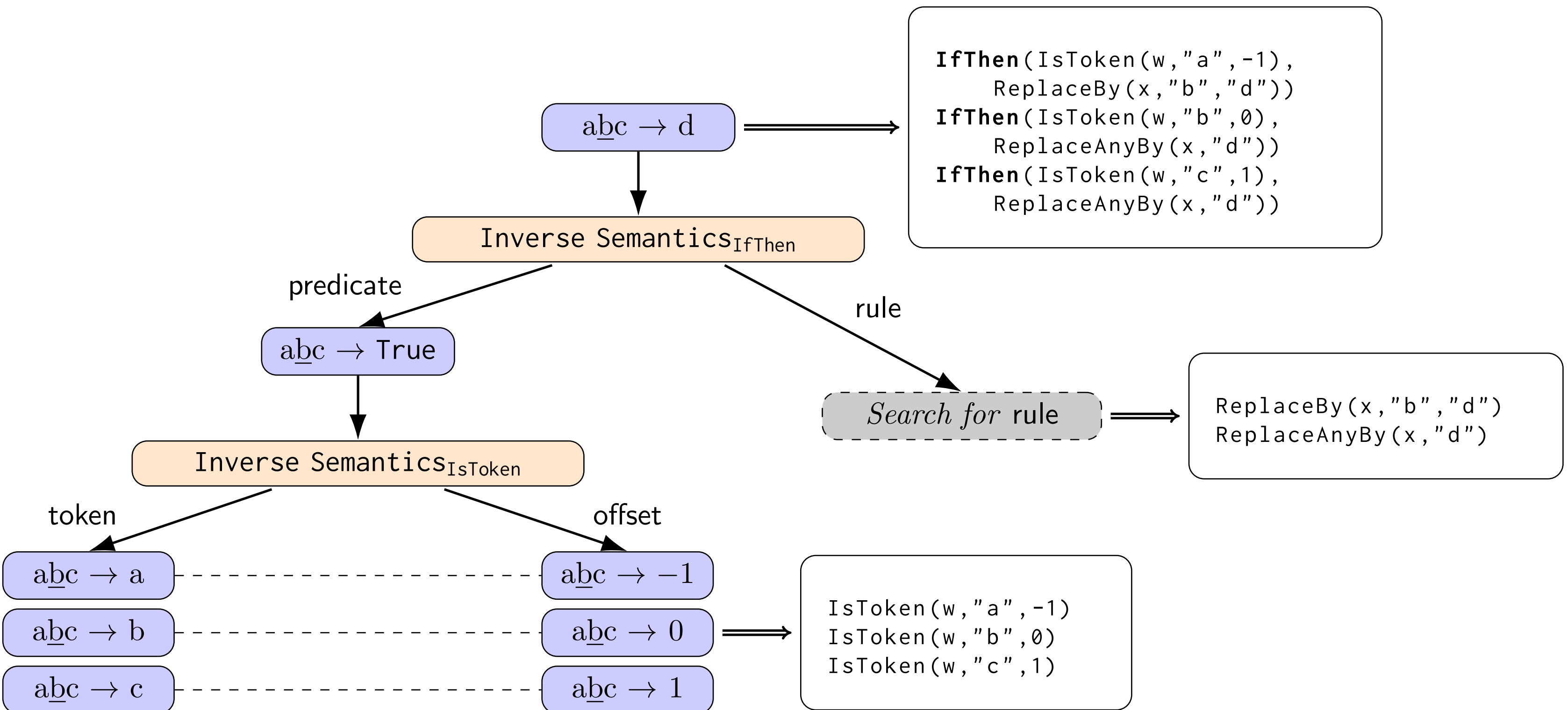

Figure 4.1: An illustration of the search performed by the FlashMeta algorithm. The blue boxes show the specification that an operator must satisfy in terms of input-output examples, with the input token underlined in the context of the word. The `Inverse Semantics` of an operator is a function that is used to infer the specification for each argument of the operator based on the semantics of the operator. This may be a single specification (as for predicate) or a disjunction of specifications (as for token and offset). The algorithm then recursively searches for programs to satisfy the specification for each argument, and combines the results of the search to obtain a program. The search for the transformation – rule – in an `IfThen` statement proceeds similarly to the search for a predicate. Examples of programs that are inferred from a specification are indicated with $\Longrightarrow$. A dashed line between inferred specifications indicates that the specifications are inferred jointly.

Chapter 5

Learning Phonological Rules as String Transformations

In this chapter, we study the following questions:

(i) Can program synthesis be used to learn linguistic rules from only a few examples?

(ii) If so, what kind of rules can be learnt?

(iii) What kind of operations need to explicitly be defined in the DSL to allow it to model linguistic rules?

(iv) What knowledge must be implicitly provided with these operations to allow the model to choose rules that generalise well?

We present a program synthesis model and a DSL for learning phonological rules and evaluate on the set of Linguistics Olympiad problems described in Chapter 2. We perform experiments and comparisons to baselines and find that program synthesis does significantly better than our baseline approaches. We also present some observations about the ability of our system to find rules that generalise well and discuss examples of where it fails.

5.1 Program synthesis

In this work, we explore learning rules similar to rewrite rules (Chomsky and Halle, 1968) that are used extensively to describe phonology. Sequences of rules are learnt using a noisy disjunctive synthesis algorithm NDSyn (Iyer et al., 2019) extended to learn *stateful multi-pass rules* (Sarthi et al., 2021).

5.1.1 Phonological rules as programs

The synthesis task we solve is to learn a program in a domain-specific language (DSL) for string transduction, that is, to transform a given sequence of input tokens $i \in \mathcal{I}^*$ to a sequence of output tokens $o \in \mathcal{O}^*$, where $\mathcal{I}$ is the set of input tokens, and $\mathcal{O}$ is the set of output tokens. Each token is a symbol accompanied by a feature set, a set of key-value pairs that maps feature names to boolean values.

We learn programs for token-level examples, which transform an input token in its context to output tokens. The program is a sequence of rules which are applied to each token in an input string to produce the output string. The rules learnt are similar to rewrite rules, of the form

$$\phi_{-l} \cdots \phi_{-2}\phi_{-1} X \phi_1 \phi_2 \cdots \phi_r \to T$$

where (i) $X : \mathcal{I} \to \mathbb{B}$ is a boolean predicate that determines input tokens to which the rule is applied (ii) $\phi_i : \mathcal{I} \to \mathbb{B}$ is a boolean predicate applied to the ith character relative to X, and the predicates ϕ collectively determine the context in which the rule is applied (iii) $T : \mathcal{I} \to \mathcal{O}^*$ is a function that maps an input token to a sequence of output tokens.

X and ϕ belong to a set of predicates $\mathcal{P}$, and T is a function belonging to a set of transformation functions $\mathcal{T}$. $\mathcal{P}$ and $\mathcal{T}$ are specified by the DSL.

We allow the model to synthesise programs that apply multiple rules to a single token by synthesising rules in passes and maintaining state from one pass to the next. This allows the system to learn stateful multi-pass rules (Sarthi et al., 2021).

5.1.2 Domain-specific language

The domain-specific language (DSL) defines the allowable string transformation operations. The DSL is defined by a set of operators, a grammar that determines how they can be combined, and a semantics that determines what each operator does. By defining operators to capture domain-specific phenomena, we can reduce the space of programs to be searched to include those programs that capture distinctions relevant to the domain. This allows us to explicitly encode knowledge of the domain into the system.

Operators in the DSL also have a score associated with each operator that allows for setting domain-specific preferences for certain kinds of programs. We can combine scores for each operator in a program to compute a ranking score that we can use to identify the most preferred program among candidates. The ranking score can capture implicit preferences like shorter programs, more/less general programs, certain classes of transformations, etc.

The DSL defines the predicates $\mathcal{P}$ and the set of transformations $\mathcal{T}$ that can be applied to a particular token. The predicates and transformations in the DSL we use, along with the description of their semantics, can be found in tables 5.1 and 5.2.

```
output := Map(disjunction, input_tokens)
disjunction := Else(rule, disjunction)
rule := transformation
        | IfThen(predicate, rule);
```

Figure 5.1: `IfThen-Else` statements in the DSL

Predicate	
`IsToken`(w, s, i)	Is x equal to the token s? This allows us to evaluate matches with specific tokens.
`Is`(w, f, i)	Is f true for x? This allows us to generalise beyond single tokens and use features that apply to multiple tokens.
`TransformationApplied`(w, t, i)	Has the transformation t been applied to x in a previous pass? This allows us to reference previous passes in learning rules for the current pass.
`Not`(p)	Negates the predicate p.

Table 5.1: Predicates that are used for synthesis. The predicates are applied to a token x that is at an offset i from the current token in the word w. The offset may be positive to refer to tokens after the current token, zero to refer to the current token, or negative to refer to tokens before the current token.

Sequences of rules are learnt as disjunctions of `IfThen` operators, and are applied to each token of the input using a `Map` operator (fig. 6.1). The conjunction of predicates X and ϕ that define the context are learnt by nesting `IfThen` operators.

A `transformation` produces a token that is tagged with the transformation that is applied. This allows for maintaining state across passes.

The operators in our DSL are quite generic and can be applied to other string transformations as well. In addition to designing our DSL for string transformation tasks, we allow for phonological information to be specified as features, which are a set of key-value pairs that map attributes to boolean values. While we restrict our investigation to features based only on the symbols in the input, more complex features based on meaning and linguistic categories can be provided to a system that works on learning rules for more complex domains like morphology or syntax. We leave this investigation for future work.

5.1.3 Synthesis algorithm

We use an extension (Sarthi et al., 2021) of the NDSyn algorithm (Iyer et al., 2019) that can synthesise stateful multi-pass rules. Iyer et al. (2019) describe an algorithm for selecting disjunctions of rules, and use the FlashMeta algorithm as the rule synthesis component. Sarthi et al. (2021) extend the approach proposed by Iyer et al. (2019) for disjunctive synthesis to the task of grapheme-to-phoneme (G2P) conversion in Hindi and Tamil. They propose the idea of learning transformations on token-aligned examples and use language-specific predicates and transformations to learn rules for G2P conversion. We use a similar approach, and use a different set of predicates and transformations that are language-agnostic. fig. 5.2 sketches the working of the algorithm.

We use the NDSyn algorithm to learn disjunctions of rules. We apply NDSyn in multiple passes to allow the model to learn multi-pass rules.

At each pass, the algorithm learns rules to perform token-level transformations that are applied to each element of the input sequence. The token-level examples are passed to NDSyn, which learns

Transformation	
ReplaceBy(x, s_1, s_2)	If x is s_1, it is replaced with s_2. This allows the system to learn conditional substitutions.
ReplaceAnyBy(x, s)	x is replaced with s. This allows the system to learn unconditional substitutions.
Insert(x, S)	This inserts a sequence of tokens S after x at the end of the pass. It allows for the insertion of variable-length affixes.
Delete(x)	This deletes x from the word at the end of the pass.
CopyReplace(x, i) CopyInsert(x, i)	These are analogues of the ReplaceBy and Insert transformations where the token which is added is the same as the token at an offset i from x. They allow the system to learn phonological transformations such as assimilation and gemination.
Identity(x)	This returns x unchanged. It allows the system where a transformation applies under certain conditions but does not under others.

Table 5.2: Transformations that are used for synthesis. The transformations are applied to a token x in the word w. The offset i for the Copy transformations may be positive to refer to tokens after the current token, zero to refer to the current token, or negative to refer to tokens before the current token.

the IfThen-Else statements that constitute a set of rules. This is done by first generating a set of candidate rules by randomly sampling a token-level example and synthesising a set of rules that satisfy the example. Then, rules are selected to cover the token-level examples.

Rules that satisfy a randomly sampled example are learnt using the FlashMeta program synthesis algorithm (Polozov and Gulwani, 2015), described in Chapter 4. After the candidates are generated, they are ranked according to the ranking score of each program. The ranking score for an operator in a program is computed as a function of the scores of its arguments. The arguments may be other operators, offsets, or other constants (like tokens or features). The score for an operator in the argument is computed recursively. The score for an offset favours smaller numbers and local rules by decreasing the score for larger offsets. The score for other constants is chosen to be a small negative constant. The scores for the arguments are added up, along with a small negative penalty to favour shorter programs, to obtain the final score for the operator.

This ranking score selects for programs that are shorter, and may favour choosing more general by giving the Is predicate a higher score or more specific rules by giving the IsToken predicate a higher score. The top k programs according to the ranking function are chosen as candidates for the next step.

To choose the final set of rules from the candidates generated using the FlashMeta algorithm, we use a *set covering* algorithm that chooses the rules that correctly answer the most number of examples while also incorrectly answering the least. These rules are applied to each example, and the output tokens are tagged with the transformation that is applied. These outputs are then the input to the next pass of the algorithm.

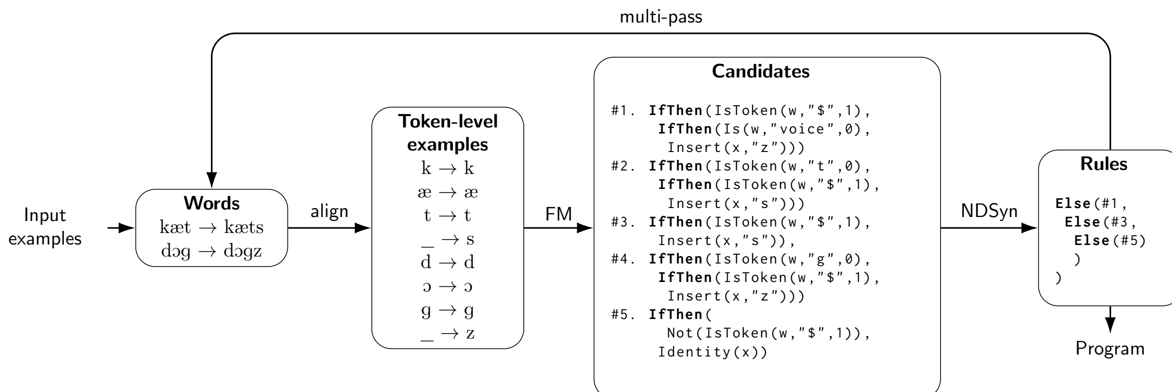

Figure 5.2: An illustration of the synthesis algorithm. FM is FlashMeta, which synthesises rules which are combined into a disjunction of rules by NDSyn. Here, rule #1 is chosen over #4 since it uses the more general concept of the `voice` feature as opposed to a specific token and thus has a higher ranking score.

The synthesis of multi-pass rules proceeds in passes. In each pass, a set of token-aligned examples is provided as input to the NDSyn algorithm. The resulting rules are then applied to all the examples, and those that are not solved are passed as the set of examples to NDSyn in the next pass. This proceeds until all the examples are solved, or for a maximum number of passes.

5.1.4 Structure of the problems

Each problem is presented in the form of a matrix M. Each row of the matrix contains data pertaining to a single word/linguistic form, and each column contains the same form of different words, i.e., an inflectional or derivational paradigm, the word form in a particular language, the word in a particular script, or the stress values for each phoneme in a word. A test sample in this case is presented as a particular cell M_{ij} in the table that has to be filled. The model has to use the data from other words in the same row ($M_{i:}$) and the words in the column ($M_{:j}$) to predict the form of the word in M_{ij}.

In addition to the data in the table, each problem contains some additional information about the symbols used to represent the words. This additional information is meant to aid the solver understand the meaning of a symbol they may not have seen before. We manually encode this information in the feature set associated with each token for synthesis. Where applicable, we also add consonant/vowel distinctions in the given features, since this is a basic distinction assumed in the solutions to many Olympiad problems.

We use the assignments that accompany every problem as the test set, ensuring that the correct answer can be inferred based on the given data.

Model	All		Morphophonology		Multilingual		Transliteration		Stress
	EXACT	CHRF	EXACT	CHRF	EXACT	CHRF	EXACT	CHRF	EXACT
NOFEATURE	26.8%	0.64	30.1%	0.72	42.1%	0.59	12.0%	0.51	15.4%
TOKEN	**32.7%**	0.63	37.5%	0.68	**45.3%**	0.60	16.4%	0.52	22.2%
FEATURE	30.9%	0.51	**38.6%**	0.56	39.9%	0.42	9.5%	0.49	23.0%
LSTM	9.7%	0.44	9.2%	0.49	5.7%	0.45	2.1%	0.31	**23.2%**
Transformer	5.3%	0.42	2.3%	0.39	9.2%	0.50	1.7%	0.42	12.1%
WFST	20.9%	0.56	16.3%	0.47	38.7%	0.63	**29.7%**	0.71	2.8%

Table 5.3: Metrics for all problems, and for problems of each type. The CHFF score for stress problems is not calculated, and not used to determine the overall CHRF score.

5.2 Experiments

5.2.1 Baselines

Given that we model our task as string transduction, we compare with the following transduction models used as baselines in shared tasks on G2P conversion (Gorman et al., 2020) and morphological reinflection (Vylomova et al., 2020).

Neural: We use LSTM-based sequence-to-sequence models with attention as well as Transformer models as implemented by Wu (2020). For each problem, we train a single neural model that takes the source and target column numbers, and the source word, and predicts the target word.

WFST: We use models similar to the pair n-gram models (Novak et al., 2016), with the implementation similar to that used by Lee et al. (2020). We train a model for each pair of columns in a problem. For each test example M_{ij}, we find the column with the smallest index j' such that $M_{ij'}$ is non-empty and use $M_{ij'}$ as the source string to infer M_{ij}.

Additional details of baselines are provided in Appendix B.

5.2.2 Program synthesis experiments

As discussed in section 5.1.4, the examples in a problem are in a matrix, and we synthesise programs to transform entries in one column into entries in another. Given a problem matrix M, we refer to a program to transform an entry in column i to an entry in column j as $M_{:i} \rightarrow M_{:j}$. To obtain token-level examples, we use the Smith-Waterman alignment algorithm (Smith et al., 1981), which favours contiguous sequences in aligned strings.

We train three variants of our synthesis system with different scores for the `Is` and `IsToken` operators. The first one, NOFEATURE, does not use features, or the `Is` predicate. The second one, TOKEN, assigns a higher score to `IsToken` and prefers more specific rules that reference tokens. The third one, FEATURE, assigns a higher score to `Is` and prefers more general rules that reference features instead of tokens. All other aspects of the model remain the same across variants.

Morphophonology and multilingual problems: For every pair of columns (s, t) in the problem matrix M, we synthesise the program $M_{:s} \rightarrow M_{:t}$. To predict the form of a test sample M_{ij}, we find a column k such that the program $M_{:k} \rightarrow M_{:j}$ has the best ranking score, and evaluate it on M_{ik}.
Transliteration problems: Given a problem matrix M, we construct a new matrix M' for each pair of columns (s, t) such that all entries in M' are in the same script. We align word pairs (M_{is}, M_{it}) using the Phonetisaurus many-to-many alignment tool (Jiampojamarn et al., 2007), and build a simple mapping f for each source token to the target token with which it is most frequently aligned. We fill in M'_{is} by applying f to each token of M_{is} and $M'_{it} = M_{it}$. We then find a program $M'_{:s} \rightarrow M'_{:t}$.
Stress problems: For these problems, we do not perform any alignment, since the training pairs are already token aligned. The synthesis system learns to transform the source string into the sequence of stress values.

5.2.3 Metrics

We calculate two metrics: exact match accuracy, and CHRF score (Popović, 2015). The exact match accuracy measures the fraction of examples the synthesis system gets fully correct.

$$\text{EXACT} = \frac{\#\{\text{correctly predicted test samples}\}}{\#\{\text{test samples}\}}$$

The CHRF score is calculated only at the token level and measures the n-gram overlaps between the predicted answer and the true answer, and allows us to measure partially correct answers. We do not calculate the CHRF score for stress problems as n-gram overlap is not a meaningful measure of performance for these problems.

5.2.4 Results

Table 5.3 summarises the results of our experiments. We report the average of each metric across problems for all problems and by category.

We find that neural models that don't have specific inductive biases for the kind of tasks we present here are not able to perform well with this amount of data. The synthesis models do better than the WFST baseline overall, and on all types of problems except transliteration. This could be due to the simple map computed from alignments before program synthesis causing errors that the rule learning process cannot correct.

5.3 Analysis

We examine two aspects of the program synthesis models we propose. The first is the way it uses the explicit knowledge in the DSL and implicit knowledge provided as the ranking score to generalise. We then consider specific examples of problems and show examples of where our models succeed and fail in learning different types of patterns.

Model	100%	$\geq 75\%$	$\geq 50\%$
NoFeature	3	5	7
Token	3	6	10
Feature	**3**	**6**	**11**
WFST	1	2	7

Table 5.4: Number of problems where the model achieves different thresholds of the Exact score.

5.3.1 Features aid generalisation

Since the test examples are chosen to test specific rules, solving more test examples correctly is indicative of the number of rules inferred correctly. In table 5.4, we see that providing the model with features allows it to infer more general rules, solving a greater fraction of more problems. We see that allowing the model to use features increases its performance, and having it prefer more general rules involving features lets it do even better.

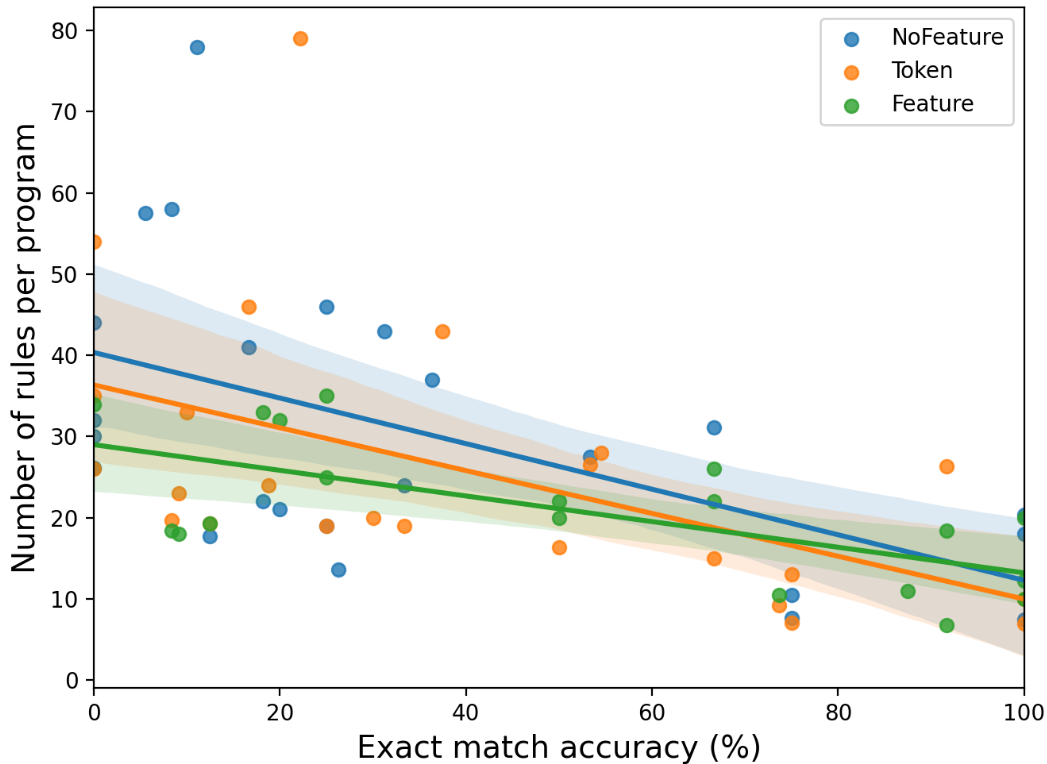

Figure 5.3: Number of rules plotted against Exact score

5.3.2 Correct programs are short

In fig. 5.3 we see that the number of rules in a problem[1] tends to be higher when the model gets the problem wrong than when it gets it right. This indicates that when the model finds many specific rules, it overfits to the training data, and fails to generalise well. This holds true for all the variants, as seen in the downward slope of the lines.

We also find that allowing and encouraging a model to use features leads to shorter programs. The average length of a program synthesised by NoFeatures is 30.5 rules, while it is 25.8 for Token, and 20.7 for Feature. This suggests that explicit access to features and implicit preference for them leads to fewer, more general rules.

5.3.3 Using features

Some problems provide additional information about certain sounds. For example, a problem based on the alternation retroflexes in Warlpiri words (Laughren, 2011) explicitly identifies retroflex sounds in the problem statement. In this case, a program produced by our Feature system is able to use these features, and isolate the focus of the problem by learning rules such as in the following program fragment:

```
IfThen(Not(Is(w, "retroflex", 0)),
    Identity(x))
```

The system learns a concise solution and is able to generalise using features rather than learning separate rules for individual sounds.

In the case of inflecting a Mandar verb (McCoy, 2018), the Feature system uses a feature to find a more general rule than is the case. To capture the rule that the prefix *di-* changes to *mas-* when the root starts with *s*, the model synthesises the following program fragment:

```
IfThen(Is(w, "fricative", 1),
    ReplaceBy(x, "i", "s"))
```

However, since *s* is the only fricative in the data, this rule is equivalent to a rule specific to *s*. This rule also covers examples where the root starts with *s*, and causes the model to miss the more general rule of a voiceless sound at the beginning of the root to be copied to the end of the prefix. It identifies this rule only for roots starting with *p* as in the following program fragment:

```
IfThen(IsToken(w, "p", 1),
    CopyReplace(x, w, 1))
```

The Token system does not synthesise these rules based on features, and instead chooses rules specific to each initial character in the root.

Since the DSL allows for substituting one token with one other, or inserting multiple tokens, the system has to use multiple rules to substitute one token with multiple tokens. In the case of Mandar, we

[1]To account for some problems having more columns than others (and hence more rules), we find the average number of rules for each pair of columns.

see one way it does this, by performing multiple substitutions (to transform *di-* to *mas-* it replaces *d* and *i* with *a* and *s* respectively, and then inserts *m*).

5.3.4 Multi-pass rules

In a problem on Somali verb forms (Somers, 2016), we see a different way of handling multi-token substitutions by using multi-pass rules to create a complex rule using simpler elements. The problem requires being able to convert verbs from 1st person to 3rd person singular. The solution includes a rule where a single token (*l*) is replaced with (*sh*). The learnt program uses two passes to capture this rule through the sequential application of two rules: first `ReplaceBy(x, "l", "h")`, followed by this program fragment in the next pass:

```
IfThen(TransformationApplied(w,
        "{ReplaceBy, h}", 1),
    Insert(x, "s"))
```

5.3.5 Selecting spans of the input

In a problem involving reduplication in Tarangan (Warner, 2019), all variants fail to capture any synthesis rules. Reduplication in Tarangan involves copying one or two syllables in the source word to produce the target word. However, the DSL we use does not have any predicates or transformations that allow the system to reference a span of multiple tokens (which would form a syllable) in the input. Therefore, it fails to model reduplication.

5.3.6 Global constraints

Since we provide the synthesis model with token-level examples, it does not have access to word-level information. This results in poor performance on stress problems, as stress depends on the entire word. Consider the example of Chickasaw stress (Vaduguru, 2019). It correctly learns the following rule:

```
IfThen(Is(w, "long", 0),
    ReplaceAnyBy(x, "1"))
```

that stresses any long vowel in the word. However, since it cannot check if the word has a long vowel that has already been stressed, it is not able to correctly model the case when the word doesn't have a long vowel. This results in some samples being marked with stress at two locations, one where the rule for long vowels applies, and one where the rule for words without long vowels applies.

5.4 Conclusion

In this chapter, we explore the problem of learning linguistic rules from only a few training examples, using a method based on program synthesis. we demonstrate that synthesis is a powerful and flexible technique for learning phonology rules in Olympiad problems. These problems are designed to be challenging tasks that require learning rules from a minimal number of examples. These problems also allow us to specifically test for generalisation.

We compare our approach to various baselines and find that it is capable of learning phonological rules that generalise much better than existing approaches. We show that using the DSL, we can explicitly control the structure of rules, and using the ranking score, we can provide the model with implicit preferences for certain kinds of rules.

Having demonstrated the potential of program synthesis as a learning technique that can work with very little data and provide human-readable models, we further explore the role of DSL design in synthesis.

Chapter 6

Learning Rules for Phonological Stress Placement

In Chapter 5, we saw 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. We demonstrated a method for learning different types of rules, including morphophonology, transliteration, and phonological stress.

In this chapter, 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 the 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 the 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. Table 6.1 shows an example of the task of phonological stress placement.

We extend the formulation of phonological stress placement as a string-to-string transformation problem from Chapter 5 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 generalisation from some given data.

6.1 Program synthesis

6.1.1 Stress rules as programs

We model stress rules as string-to-string transformations. Formally, we synthesise a program that implements a function $f : \Sigma^* \to \{0, 1, 2, 3\}^*$, where Σ is the set of phonemes in a language. f takes as input a sequence of phonemes $w_1 w_2 \ldots w_n$, 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

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

Table 6.1: 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.

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 synthesise take the form of sequences of rules similar to rewrite rules Chomsky and Halle (1968).

Each rule is of the form

$$\phi_{-l} \cdots \phi_{-1} X_1 \cdots X_c \phi_1 \cdots \phi_r \rightarrow T \tag{6.1}$$

A rule applies to a central phoneme that satisfies a conjunction of predicates $X_1, \ldots, X_c$, which appears in a context defined by the conjunction of predicates $\phi_{-l}, \ldots, \phi_{-1}$ (which apply to l phonemes to the left of the central phoneme) and $\phi_1, \ldots, \phi_r$ (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 the set of transformations $\mathcal{T}$ that can be used. We vary the predicates ($\mathcal{P}$) available to the synthesiser to define different DSLs, each providing access to different classes of phonological abstractions. The transformation is a function that takes the phoneme as input and outputs the degree of stress. It is of the form `ReplaceBy(`s`)`, where s is a value representing the degree of stress.

6.1.2 Domain-specific languages

We use a DSL that implements rules of the form described in Section 6.1.1 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 6.1.1, 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 6.1: `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.

6.1.2.1 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 synthesiser 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 synthesiser, 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 6.2.

6.1.2.2 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 these 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 occurrence 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. (6.1)), or phonemes in the context (one of the ϕ_i from eq. (6.1)). We guide the synthesiser to prefer simpler rules by ranking predicates that refer to nearby phonemes (at a smaller displacement from the central

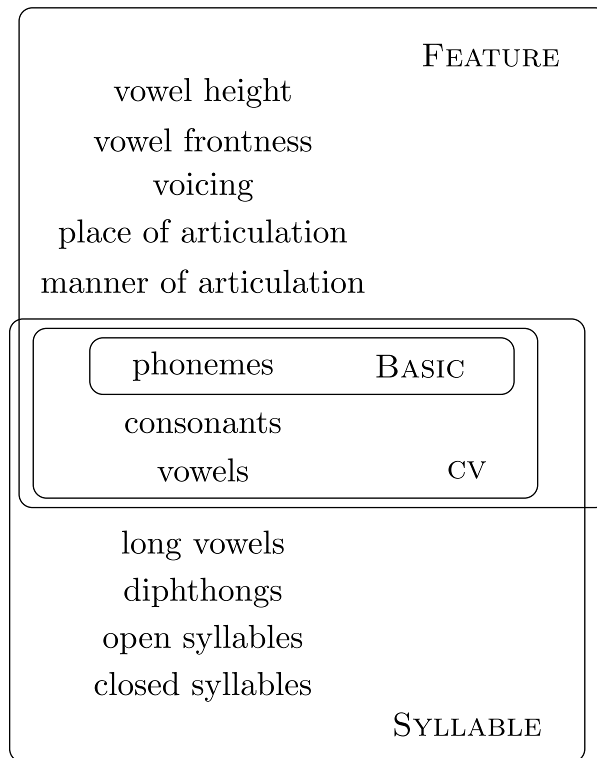

Figure 6.2: Classes available to each DSL – BASIC, CV, SYLLABLE, and FEATURE.

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 synthesiser 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 including, 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.

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

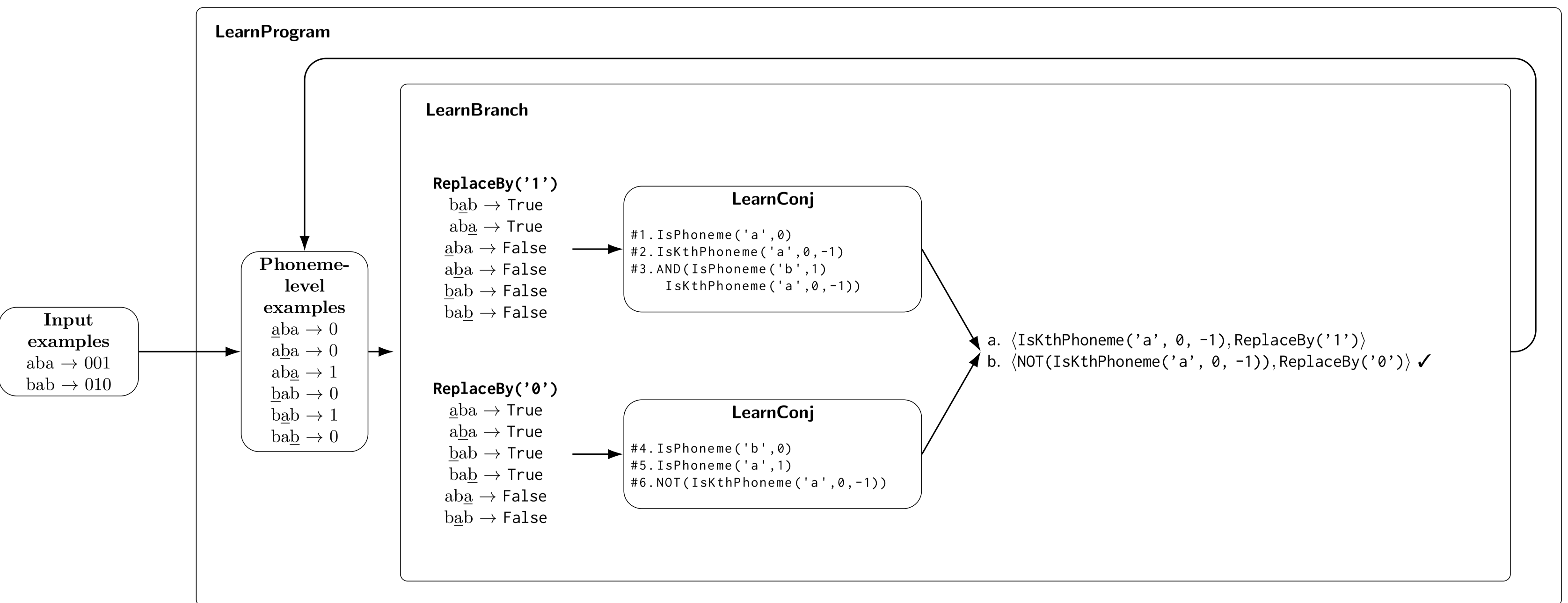

Figure 6.3: Illustration of the synthesis algorithm on a hypothetical case where the stress is on the last vowel, using the BASIC DSL. The input examples are first used to generate phoneme-level examples. The LearnProgram procedure then learns a decision list for the phoneme-level examples through calls 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.

6.1.3 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 synthesise `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, optimising for a shorter list. The procedure maintains a set R of examples that haven't yet been covered by any of the predicates of the decision list, which is initialised 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 that satisfy the predicate are then removed 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 synthesised using the FlashMeta algorithm. Given the transformation or predicate operator (as described in Section 6.1.2), 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. (6.1), 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 6.3 illustrates the working of the synthesis algorithm.

6.2 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 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. Further details about the languages are presented in Appendix A.

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 synthesising programs (the *training* split) and the other (the *test* split) to be used for evaluating the synthesised 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 chosen from the Linguistics Olympiads, which were described in Chapter 2.

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.

6.3 Experiments

As described in Section 6.2, 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 generalise 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 languages, synthesis terminates well before this limit. For each language, we experiment with each of the 4 DSLs described in Section 6.1.2.

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 Chapter 5.

To evaluate the synthesised 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.

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 6.2: 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 6.3: Number of languages where the system obtains perfect test accuracy, or test accuracy over 50%.

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

6.3.1 Results

The results obtained are shown in Tables 6.2 and 6.3. 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 larger gains relative to access to only consonant-vowel distinctions.

6.4 Analysis

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

6.4.1 Benefits of the consonant-vowel distinction

We see that providing the synthesiser access to the distinction between vowels and consonants can improve its performance significantly. A synthesiser 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 synthesiser 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 in the following program fragment:

```
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 following rule:

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

This 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 generalise well.

On the other hand, with the CV DSL, the system just learns the following rules:

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

These 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.

6.4.2 Benefits from syllable-level distinctions

The benefits of being able to refer to syllable-level information in rules is visible in the programs synthesised 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 the following:

```
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 generalise 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.

6.4.3 Incorrect generalisations

However, providing access to syllable-level distinctions may also encourage the synthesiser to discover incorrect generalisations. 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 following rule:

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

This 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.

6.4.4 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 synthesised 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 synthesised:

```
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 is sufficiently general. They 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 prevents 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 synthesised with the SYLLABLE DSL for the same data includes the following 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 should receive primary stress.

6.5 Related work

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.

6.6 Conclusion

In this chapter, 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 synthesised, we can control the amount of linguistic information available to the synthesiser.

We use the allowance to explicitly 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 synthesised. 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 synthesiser 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 the 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.

Chapter 7

Conclusions

In this thesis, we have designed methods that can learn linguistic rules from a small number of examples and evaluated their efficacy on a set of phonology problems curated from Linguistics Olympiads. We also studied the effect of prior knowledge on how the learned rules generalise to unseen examples in a case study of learning rules for phonological stress placement.

This task of learning rules can be viewed from two perspectives, as discussed in Chapter 3 – learning phonology and morphology from limited data and learning rules for abstract reasoning tasks.

From the perspective of learning phonology and morphology from limited data, we have shown that program synthesis can be a method to learn aspects of phonology and morphology from very small amounts of data. In addition to the system being highly data-efficient, the resulting rules are in the form of human-readable programs and the ways in which the model generalises can be controlled. This controllable nature also lends itself to using program synthesis to study the types of learning biases required to model phonological processes.

While the fully symbolic nature of the models we discuss allows for control and generalisation, integrating data-driven learning will allow for leveraging more data when available, taking into account factors beyond the word forms in the language such as typology, and also adapting better to variation and irregular forms in natural language with statistical learning.

From the perspective of learning rules for abstract reasoning, the Olympiad problems continue to pose a significant challenge. The systems we propose are able to tackle only specific types of problems from the Olympiad, and we remain far from being able to tackle the task in its most general form – where problems may involve reasoning about intricate syntactic structures, and complex semantic phenomena.

Johnson et al. (2021) note one feature of the ARC challenge that is also present in Olympiad problems, but one that we do not account for in our modeling – the aspect of *abductive* reasoning as opposed to inductive reasoning. Solving Olympiad problems requires being able to flexibly hypothesise new abstract concepts "on-the-fly", and correctly and consistently bind values to these variables to solve the task. They speculate that one possible way humans generate hypotheses is by using natural language as a scaffold.

Acquaviva et al. (2021) explore this aspect of the problem, using a communication game to collect natural language data about strategies to solve the ARC challenge. They identify program-like elements in natural language instructions on solving the tasks and find that using natural language to guide the search for programs improves performance significantly. The advantage of using natural language to guide the discovery of programmatic abstractions has also been demonstrated by Wong et al. (2021) and Andreas et al. (2018). Leveraging natural language instructions to learn about how human solvers hypothesise abstract concepts is a promising direction in informing the search for program-like solutions to complex abstract reasoning tasks such as solving Linguistics Olympiad problems.

Appendix A

Data from grammars and other linguistic and phonological analyses of languages listed in the STRESSTYP2 database Goedemans et al. (2014) or by Gordon (2002) were used to create the dataset. The STRESSTYP2 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 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. All languages used are listed in Table A.1.

A.1 Olympiad data

Language-wise results are listed in Table A.2.

A.2 Examples

Examples of problems are provided in Table A.3 and Table A.4.

Language	Source	BASIC	CV	SYLLABLE	FEATURE	LSTM	Transformer
Anejom	Lynch, 2000	0	0	0.666667	0	0	0
Arabic	Wright, 1874	0.25	0.75	0.25	0.5	25	50
Araucanian	Echeverría and Contreras, 1965	0	0.666667	0.333333	0.333333	0	0
Au	Scorza, 1985	0	0.666667	0.333333	0	33.3333	0
Awtuw	Feldman, 1983	0.333333	1	1	1	0	0
Bortzerrieta Basque	Hualde, 1989	0	1	1	1	0	0
Cofan	Borman, 1962	0	0.333333	0.666667	0.666667	33.3333	0
Coreguaje	Gralow, 1985	0	1	1	1	50	50
Garawa	Furby, 1974	0	0	1	1	0	0
Hopi	Jeanne, 1978	0.4	0.4	0.8	0.6	0	20
Kaliai-Kove	Counts, 1969	0.25	1	1	1	25	0
Kara	Schlie and Schlie, 1993	0.25	0.25	0	0	0	0
Kwakiutl	Bach, 1975	0.2	0	0.2	0	0	40
Lezgian	Haspelmath, 1993	0	1	1	1	50	0
Manam	Buckley, 1998	0	0.25	1	0.25	0	0
Mathimathi	Hercus, 1986	0	0.25	0	0	0	25
Murik	Abbott, 1985	0.5	0.75	0.75	0.75	0	0
Piro	Matteson, 1965	0	0	0	0	0	0
Sarangani Monobo	Dubois, 1976	0.5	0.5	0.75	1	0	0
Seri	Marlett, 1988	0.75	0.75	0.75	0.75	50	50
Sio	Soetenga Clark, 1993	0.5	0.5	1	0.75	0	0
Tahitian	Tryon, 1976	0.666667	0.666667	0.666667	0.666667	0	66.6667
Tamazight	Abdel-Massih, 1971	0.666667	1	1	1	0	0
Tinrin	Osumi, 1995	0	1	1	1	0	0
Tol	Fleming and Dennis, 1977	0	0.25	1	1	50	25
Tumpisa	Dayley, 1989	0	0	0.333333	0.333333	33.3333	33.3333
Tzutujil	Dayley, 1985	0	0.8	0.4	0.4	20	0
Yidiny	Dixon, 1977	0	0	0	0	0	0

Table A.1: Languages for which we collect data.

Problem Name	BASIC	CV	SYLLABLE	FEATURE	LSTM	Transformer
Aliutor	0.25	0	0.4375	0.4375	37.5	31.25
Chickasaw Stress	0	0	0.833333	0.166667	0	0
Creek	0.416667	0	0.5	0.083333	33.3333	0
Kabardian	0.5	1	1	1	33.3333	16.6667
Old Indic	0	0	0	0.25	25	25
Piraha	0	0	0	0	10	0

Table A.2: Olympiad problems. Kabardian is an IOL problem in which stress is presented as a side phenomenon, and is easier to deduce than in other IOL problems.

Creek word	Stress pattern
coko	0001
ca:lo	0100
sokca	01000
wa:koci	000001
pocoswa	0001000
fami:ca	000100
yanasa	000100
iyanawa	0000001
hî:spakwa	00001000
aklowahi:	00000001
imahicita	000000100
inkosapita	0000000001
tapasso:la	000000100
akkopanka	000001000
cokpilâ:pila	00000000001
tokna:photi	0000000001
co:kakiłłita	00000000001
łafotahaya	0000000100
itiwanayipita	0000000000100
ipahankatita	000000000100
pokkołakkoakkopankacoko	0000000000000000000100
ifa	?
nâ:naki	?
aktopa	?
wanayita	?
isiskitoci	?
honanta:ki	?
ifoci	?
sâ:sakwa	?
hoktaki	?
a:tamihoma	?
awanayita	?
ilitohtałita	?

Table A.3: Creek stress

Kabardian word	**Stress pattern**
defən	00010
gʷəš’əʔen	0000010
made	0100
mašxe	01000
mec̣antχʷe	00010000
megʷəš’əʔef	000000010
mes	010
mebəbəpe	00000100
mešxape	0000100
meʒeq̇afe	00000100
sən	010
ṭəgʷərəgʷən	000000010
ʒeq̇en	?
medef	?
medafe	?
səfən	?
megʷəš’əʔe	?
mebəb	?
c̣antχʷeq̇əm	010000000
c̣antχʷet	0100000
c̣entχʷéft	000t0100
dapet	01000
defxeme	0100000
c̣antχʷexeme	0100000000
meṭəgʷərəgʷxe	00000001000
səfq̇əm	010000
bəbme	01000
bəbxet	010000
šxeme	00100
ṭəgʷərəgʷq̇əm	0000010000
c̣entχʷefme	?
šxafexeq̇əm	?
bəbəft	?
šxet	?
ṭəgʷərəgʷəpeme	?

Table A.4: Kabardian stress

Appendix B

B.1 Neural

Following Şahin et al. (2020), we use small neural models for sequence-to-sequence tasks. We train a single neural model for each task, and provide the column numbers as tags in addition to the source sequence. We find that the single model approach works better than training a model for each pair of columns.

LSTM: We use LSTM models with soft attention Luong et al. (2015), with embeddings of size 64, hidden layers of size 128, a 2-layer encoder and a single layer decoder. We apply a dropout of 0.3 for all layers. We train the model for 100 epochs using the Adam optimiser with a learning rate of 10^{-3}, learning rate reduction on plateau, and a batch size of 2. We clip the gradient norm to 5.

Transformer: We use Transformer models Vaswani et al. (2017) with embeddings of size 128, hidden layers of size 256, a 2-layer encoder and a 2-layer decoder. We apply a dropout of 0.3 for all layers. We train the model for 2000 steps using the Adam optimiser with a learning rate of 10^{-3}, warmup of 400 steps, learning rate reduction on plateau, and a batch size of 2. We use a label smoothing value of 0.1, and clip the gradient norm to 1.

We use the implementations provided at `https://github.com/shijie-wu/neural-transducer/` for all neural models.

B.2 WFST

We use the implementation the WFST models available at `https://github.com/sigmorphon/2020/tree/master/task1/baselines/fst` for the WFST models. We train a model for each pair of columns. We report the results for models of order 5, which were found to perform the best on the test data (highest EXACT score) among models of order 3 to 9.

Related Publications

1. **Saujas Vaduguru**, Partho Sarthi, Monojit Choudhury, and Dipti Sharma. *Stress Rules from Surface Forms: Experiments with Program Synthesis*. In *International Conference on Natural Language Processing (ICON)*, 2021. Accepted.

2. **Saujas Vaduguru**, Aalok Sathe, Monojit Choudhury, and Dipti Sharma. *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*, 2021. Accepted.

Bibliography

Stan Abbott. 1985. *A Tentative Multilevel Multiunit Phonological Analysis of the Murik Language*, volume 63 of *Pacific Linguistics, Series A*. Australian National University, Canberra.

E.T. Abdel-Massih. 1971. *A Reference Grammar of Tamazight: A Comparative Study of the Berber Dialects of Ayt Ayache and Ayt Seghrouchen*. Number v. 1 in Publications of the Center for Near Eastern and North African Studies, University of Michigan. Center for Near Eastern and North African Studies, University of Michigan.

Samuel Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle E Ecanow, Maxwell Nye, Michael Henry Tessler, and Joshua B. Tenenbaum. 2021. Communicating natural programs to humans and machines.

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. 2021. Learning to recombine and resample data for compositional generalization. In *International Conference on Learning Representations*.

Antonios Anastasopoulos and Graham Neubig. 2019. Pushing the limits of low-resource morphological inflection. 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 984–996, Hong Kong, China. Association for Computational Linguistics.

Jacob Andreas. 2020. Good-enough compositional data augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7556–7566, Online. Association for Computational Linguistics.

Jacob Andreas, Dan Klein, and Sergey Levine. 2018. Learning with latent language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2166–2179, New Orleans, Louisiana. Association for Computational Linguistics.

E. Bach. 1975. Long vowels and stress in kwakiutl. *Texas Linguistic Forum*, 2:9–19.

Sorav Bansal and Alexander Aiken. 2008. Binary translation using peephole superoptimizers. In *OSDI*.

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.

A. Bellos. 2020. *The Language Lover's Puzzle Book: Lexical perplexities and cracking conundrums from across the globe*. Guardian Faber Publishing.

M. B. Borman. 1962. Cofan phonemes. In Benjamin F. Elson, editor, *Studies in Ecuadorian Indian languages 1*, volume 7 of *Linguistic Series*, pages 45–59. Summer Institute of Linguistics of the University of Oklahoma, Norman.

Bozhidar Bozhanov and Ivan Derzhanski. 2013. Rosetta stone linguistic problems. In *Proceedings of the Fourth Workshop on Teaching NLP and CL*, pages 1–8, Sofia, Bulgaria. Association for Computational Linguistics.

Eric Brill. 1992. A simple rule-based part of speech tagger. In *Third Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy. Association for Computational Linguistics.

Keith Brown and Sarah Ogilvie. 2010. *Concise encyclopedia of languages of the world*. Elsevier.

Eugene Buckley. 1998. Alignment in manam stress. *Linguistic Inquiry*, 29(3):475–496.

Aditi Chaudhary, Antonios Anastasopoulos, Adithya Pratapa, David R. Mortensen, Zaid Sheikh, Yulia Tsvetkov, and Graham Neubig. 2020. Automatic extraction of rules governing morphological agreement. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5212–5236, Online. Association for Computational Linguistics.

François Chollet. 2019. On the measure of intelligence.

N. Chomsky. 1975. *Reflections on Language*. Pantheon Books. Pantheon Books.

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

David R. Counts. 1969. A grammar of kaliai-kove. *Oceanic Linguistics Special Publications*, (6):i–170.

Jon P. Dayley. 1985. *Tzutujil Grammar*, volume 107 of *University of California Publications in Linguistics*. University of California Press, Berkeley and Los Angeles. Bibliography: p. 409-412.

Jon P. Dayley. 1989. *Tümpisa (Panamint) Shoshone Grammar*, volume 115 of *University of California Publications in Linguistics*. University of California Press, Berkeley.

Ivan Derzhanski and Thomas Payne. 2010. *The Linguistic Olympiads: academic competitions in linguistics for secondary school students*, page 213–226. Cambridge University Press.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 345–356, New York, NY, USA. Association for Computing Machinery.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR.

Robert M. W. Dixon. 1977. Some phonological rules in yidin. *Linguistic Inquiry*, 8:1–34.

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

Carl D. Dubois. 1976. *Sarangani Manobo: An Introductory Guide*, volume 6 of *Special monograph issue*. Linguistic Society of the Philippines, Manila.

Max S. Echeverría and Heles Contreras. 1965. Araucanian phonemics. *International Journal of American Linguistics*, 31(2):132–135.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2015. Unsupervised learning by program synthesis. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 973–981. Curran Associates, Inc.

H. Feldman. 1983. *A Grammar of Awtuw*. Australian National University.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, Online. Association for Computational Linguistics.

Ilah Fleming and Ronald K. Dennis. 1977. Tol (jicaque): Phonology. *International Journal of American Linguistics*, 43(2):121–127.

Christine E. Furby. 1974. *Garawa Phonology*, volume 37 of *Pacific Linguistics, Series A*, pages 1–11. Australian National University, Canberra.

Daniel Gildea and Daniel Jurafsky. 1996. Learning bias and phonological-rule induction. *Computational Linguistics*, 22(4):497–530.

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.

Kyle Gorman, Lucas F.E. Ashby, Aaron Goyzueta, Arya McCarthy, Shijie Wu, and Daniel You. 2020. The SIGMORPHON 2020 shared task on multilingual grapheme-to-phoneme conversion. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 40–50, Online. Association for Computational Linguistics.

Frances L. Gralow. 1985. Coreguaje: Tone, stress and intonation. In Ruth M. Brend, editor, *From phonology to discourse: Studies in six Colombian languages*, volume 9 of *Language Data, Amerindian Series*, pages 3–11. Summer Institute of Linguistics, Dallas.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330.

Sumit Gulwani and Prateek Jain. 2017. Programming by examples: Pl meets ml. In *APLAS 2017*. Springer.

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.

Martin Haspelmath. 1993. *A Grammar of Lezgian*, volume 9 of *Mouton Grammar Library*. Mouton de Gruyter, Berlin.

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.

Luise. A. Hercus. 1986. Outline of the madimadi language. In L. A. Hercus, editor, *Victorian Languages: A Late Survey*, volume 77 of *Pacific Linguistics, Series B*, pages 101–151. Australian National University, Canberra.

José Ignacio Hualde. 1989. A lexical phonology of basque.

Arun Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram K Rajamani. 2019. Synthesis and machine learning for heterogeneous extraction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 301–315.

L. Jeanne. 1978. Aspects of hopi grammar.

Sittichai Jiampojamarn, Grzegorz Kondrak, and Tarek Sherif. 2007. Applying many-to-many alignments and hidden Markov models to letter-to-phoneme conversion. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 372–379, Rochester, New York. Association for Computational Linguistics.

Aysja Johnson, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. 2021. Fast and flexible: Human program induction in abstract reasoning tasks. *ArXiv*, abs/2103.05823.

Mark Johnson. 1984. A discovery procedure for certain phonological rules. In *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*, pages 344–347, Stanford, California, USA. Association for Computational Linguistics.

Vladimir Jojic, Sumit Gulwani, and Nebojsa Jojic. 2006. Probabilistic inference of programs from input/output examples. Microsoft Research.

Katharina Kann and Hinrich Schütze. 2018. Neural transductive learning and beyond: Morphological generation in the minimal-resource setting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3254–3264, Brussels, Belgium. 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.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2873–2882. PMLR.

Brenden M Lake. 2019. Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Brenden M. Lake, Tal Linzen, and Marco Baroni. 2019. Human few-shot learning of compositional instructions.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. 2017. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40:e253.

Mary Laughren. 2011. Stopping and flapping in warlpiri. In Dragomir Radev and Patrick Littell, editors, *North American Computational Linguistics Olympiad 2011: Invitational Round*. North American Computational Linguistics Olympiad.

Jackson L. Lee, Lucas F.E. Ashby, M. Elizabeth Garza, Yeonju Lee-Sikka, Sean Miller, Alan Wong, Arya D. McCarthy, and Kyle Gorman. 2020. Massively multilingual pronunciation modeling with WikiPron. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4223–4228, Marseille, France. European Language Resources Association.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

John Lynch. 2000. *A Grammar of Anejom*, volume 507 of *Pacific Linguistics*. Australian National University, Canberra.

Stephen A. Marlett. 1988. The syllable structure of seri. *International Journal of American Linguistics*, 54(3):245–278.

Esther Matteson. 1965. *The Piro (Arawakan) Language*. University of California Press, Berkeley.

Tom McCoy. 2018. Better left unsaid. In Patrick Littell, Tom McCoy, Dragomir Radev, and Ali Sharman, editors, *North American Computational Linguistics Olympiad 2018: Invitational Round*. North American Computational Linguistics Olympiad.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Tauman Kalai. 2013. A machine learning framework for programming by example. In *ICML*.

Elliott Moreton, Joe Pater, and Katya Pertsova. 2017. Phonological concept learning. *Cognitive science*, 41 1:4–69.

Josef Robert Novak, Nobuaki Minematsu, and Keikichi Hirose. 2016. Phonetisaurus: Exploring grapheme-to-phoneme conversion with joint n-gram models in the wfst framework. *Natural Language Engineering*, 22(6):907–938.

Maxwell Nye, Armando Solar-Lezama, Josh Tenenbaum, and Brenden M Lake. 2020. Learning compositional rules via neural program synthesis. In *Advances in Neural Information Processing Systems*, volume 33, pages 10832–10842. Curran Associates, Inc.

Midori Osumi. 1995. Tinrin grammar. *Oceanic Linguistics Special Publications*, (25):i–304.

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

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Gözde Gül Şahin, Yova Kementchedjhieva, Phillip Rust, and Iryna Gurevych. 2020. PuzzLing Machines: A Challenge on Learning From Small Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1241–1254, Online. Association for Computational Linguistics.

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.*

Eric Schkufza, Rahul Sharma, and Alexander Aiken. 2013. Stochastic superoptimization. *ArXiv*, abs/1211.0557.

P. Schlie and G. Schlie. 1993. A kara phonology. In J. Clifton, editor, *Phonologies of Austronesian Languages 2*, pages 99–130. Summer Institute of Linguistics, Ukarumpa.

David Scorza. 1985. *A Sketch of Au Morphology and Syntax*, volume 63 of *Pacific Linguistics, Series A*. Australian National University, Canberra.

Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

D. Soetenga Clark. 1993. The phonology of the sio language. In J. Clifton, editor, *no booktitle*, volume 2 of *Phonologies of Austronesian Languages*, pages 25–70. Summer Institute of Linguistics, Ukarumpa.

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

Harold Somers. 2016. Changing the subject. In Andrew Lamont and Dragomir Radev, editors, *North American Computational Linguistics Olympiad 2016: Invitational Round*. North American Computational Linguistics Olympiad.

Darrell T. Tryon. 1976. *Conversational Tahitian: an introduction to the Tahitian Language of French Polynesia*. Australian National University Press, Canberra.

Saujas Vaduguru. 2019. Chickasaw stress. In Shardul Chiplunkar and Saujas Vaduguru, editors, *Panini Linguistics Olympiad 2019*. Panini Linguistics Olympiad.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Ekaterina Vylomova, Jennifer White, Elizabeth Salesky, Sabrina J. Mielke, Shijie Wu, Edoardo Maria Ponti, Rowan Hall Maudslay, Ran Zmigrod, Josef Valvoda, Svetlana Toldova, Francis Tyers,

Elena Klyachko, Ilya Yegorov, Natalia Krizhanovsky, Paula Czarnowska, Irene Nikkarinen, Andrew Krizhanovsky, Tiago Pimentel, Lucas Torroba Hennigen, Christo Kirov, Garrett Nicolai, Adina Williams, Antonios Anastasopoulos, Hilaria Cruz, Eleanor Chodroff, Ryan Cotterell, Miikka Silfverberg, and Mans Hulden. 2020. SIGMORPHON 2020 shared task 0: Typologically diverse morphological inflection. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 1–39, Online. Association for Computational Linguistics.

Elysia Warner. 2019. Tarangan. In Samuel Ahmed, Bozhidar Bozhanov, Ivan Derzhanski (technical editor), Hugh Dobbs, Dmitry Gerasimov, Shinjini Ghosh, Ksenia Gilyarova, Stanislav Gurevich, Gabrijela Hladnik, Boris Iomdin, Bruno L'Astorina, Tae Hun Lee (editor-in chief), Tom McCoy, André Nikulin, Miina Norvik, Tung-Le Pan, Aleksejs Peguševs, Alexander Piperski, Maria Rubinstein, Daniel Rucki, Artūrs Semeņuks, Nathan Somers, Milena Veneva, and Elysia Warner, editors, *International Linguistics Olympiad 2019*. International Linguistics Olympiad.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. *31st IEEE International Conference on Software Engineering*, pages 364–374.

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. 2021. Leveraging language to learn program abstractions and search heuristics. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11193–11204. PMLR.

William Wright. 1874. *A Grammar of the Arabic Language*, 3 edition. Librairie du Liban, Beirut.

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

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

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal neural program synthesis from multimodal specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 1691–1704, Punta Cana, Dominican Republic. Association for Computational Linguistics.