AInix: An open platform for natural language interfaces to shell commands

Turing Scholars Undergraduate Honors Thesis
University of Texas at Austin

David Gros

Supervised By:
Dr. Raymond Mooney

Second Reader: Dr. Greg Durrett
Departmental Reader: Dr. Robert van de Geijn

May 2019

Contents

1 Abstract **3**

2 Introduction **4**

3 Background and Motivation **6**

3.1 Relevant Terms and Concepts . . . 6

3.2 Currently Available Platforms and Tools . . . 6

3.2.1 Digital Assistant Platforms . . . 6

3.2.2 Semantic Parsing Frameworks . . . 7

3.2.3 Previous Natural Language to Shell Commands Work . . . 7

3.3 Why Unix Commands? . . . 8

3.3.1 Brief History of the Unix Shell Commands . . . 8

4 Platform Architecture **9**

4.1 aish . . . 10

5 AInix Kernel Dataset **10**

5.1 Key Features . . . 10

5.1.1 Many-to-many . . . 10

5.1.2 Replacers . . . 12

5.2 The Arche Release . . . 13

6 AInix Grammar Files **14**

6.1 Challenges of Parsing Unix Commands . . . 16

6.2 Terminology . . . 16

6.3 Enabling Flexible Parsing . . . 17

6.4 Limitations . . . 18

7 Explored Models **19**

7.1 Seq2Seq . . . 19

7.2 Grammar Guided Methods . . . 20

7.3 Retrieval Based Methods . . . 20

7.3.1 UNIX-focused contextual embedding . . . 21

7.3.2 Background . . . 21

7.3.3 Collecting a code-focused dataset . . . 21

7.3.4 CookieMonster Model . . . 22

7.3.4.1 CookieMonster Tokenization Scheme . . . 22

7.3.4.2 CookieMonster Architecture . . . 25

7.3.5 Nearest Neighbor Method . . . 25

8 Results and Discussion **26**
8.1 Seq2Seq Results 27
8.2 Grammar Guided Model Results 27
8.3 Nearest Neighbor Model Results 27
8.4 Effects of Dataset Decisions 28

9 Future Work **29**

10 Conclusion **30**

11 Acknowledgements **31**

1 Abstract

This report discusses initial work on the AInix Platform. This platform is designed to allow developers to add natural language interfaces to Unix-like shell commands. This can be used with the aish shell, which allows users to intermix natural language with shell commands. We create a high-level way of specifying semantic parsing grammars and collect a dataset of basic shell commands. We experiment with seq2seq models, abstract syntax networks (ASN), and embedded nearest neighbor-based models. We find highest accuracy is achieved with seq2seq models and ASN's. While not as accurate, we find that when embedders are pretrained on large-scale code-related text, nearest neighbor models can achieve decent performance.

2 Introduction

In this report we discuss work on creating the AInix platform, a free and open-source platform designed to both:

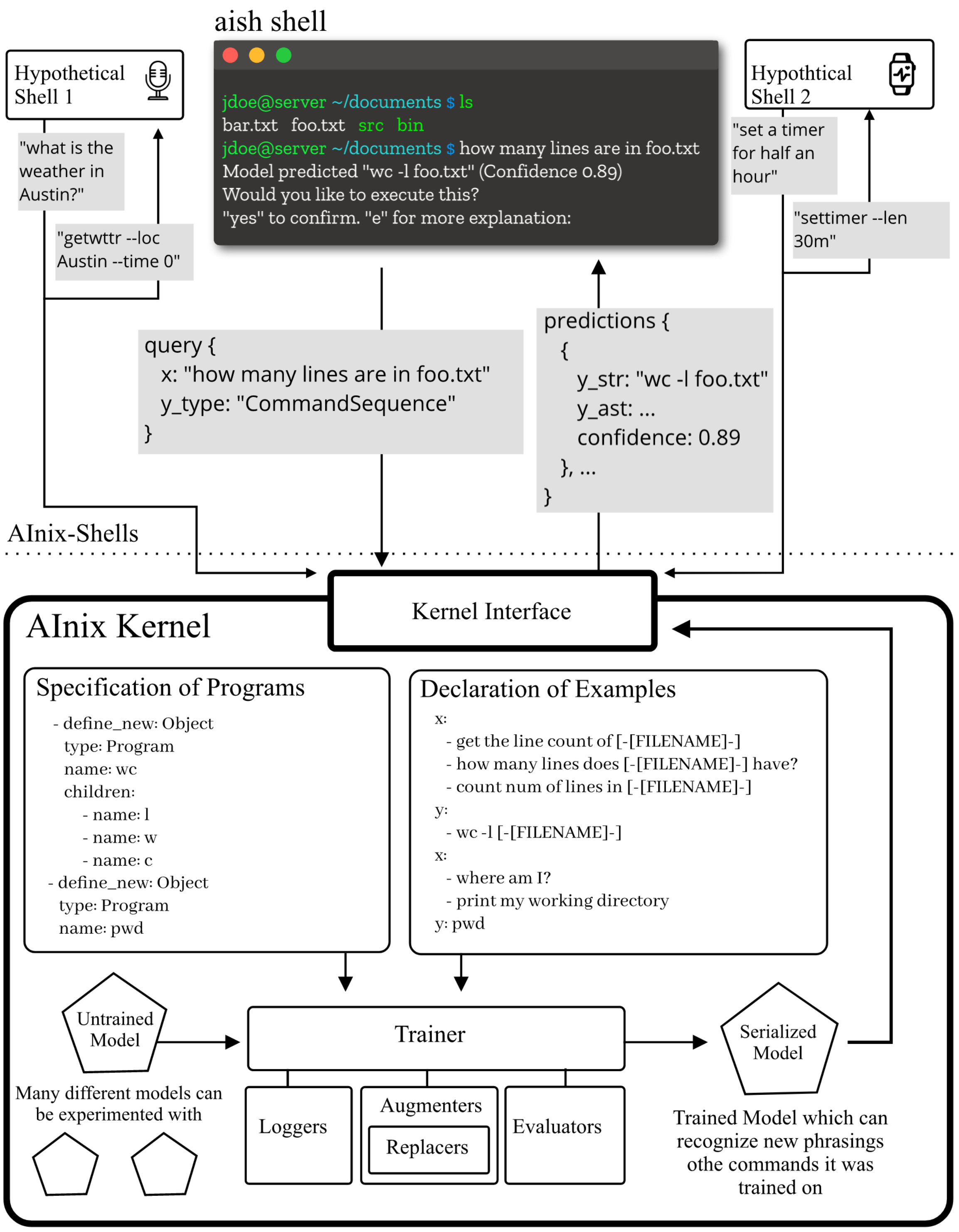
1. Enable software developers to easily create natural language and intelligent interfaces for computing tasks.

2. Allow users to benefit from these interfaces in ways that fit into existing workflows and infrastructure.

The initial focus is allowing users to complete tasks with core Unix-like shell commands such as "ls", "cp", "wc" using natural language, thus making it easier to use these commands. Such commands are used for listing files on a computer, copying files on a computer, or calculating statistics on files such as the number of lines or words. Users of these commands often have to memorize a large set of commands and options or frequently must refer to online or offline references.

While the initial scope and focus of this report is traditional Unix commands, the platform is designed from the beginning to be able to be eventually extended to a diverse range of tasks like checking the weather, purchasing something online, or interacting with Internet of Things (IoT) devices, and be able to support varying interfaces such as voice assistants, chatbots, or video game characters.

In this report, we first motivate the goals of the platform and present an argument why our chosen roadmap seems viable to meet these goals (section 3). We then elaborate more on the architecture of the platform, and how it is designed to meet these goals (section 4). We then move onto more specific implementation details. We detail the dataset that we use to train the ML models for the platform (section 5). The dataset was designed with features specialized to the needs of the project. In addition, we detail the way AInix enables simple specification of grammars for structured prediction (section 6). We then introduce three different kinds of machine learning models (seq2seq, abstract syntax tree networks, and nearest neighbor models) which we experiment with (section 7). Finally, we discuss experiment results and evaluate which kinds of models are best for the platform (section 8). We find that abstract syntax tree networks do not perform significantly worse or better than seq2seq models. We also find the nearest neighbor techniques we explored do not perform as accurately, but provide as a promising path forward.


aish shell
Hypothetical Shell 1
"what is the weather in Austin?"
"getwttr --loc Austin --time 0"
Hypothtical Shell 2
"set a timer for half an hour"
"settimer --len 30m"
AInix-Shells
Kernel Interface
AInix Kernel
Untrained Model
Many different models can be experimented with
Trainer
Loggers
Augmenters
Replacers
Evaluators
Serialized Model
Trained Model which can recognize new phrasings othe the commands it was trained on

```
jdoe@server ~/documents $ ls
bar.txt   foo.txt   src   bin
jdoe@server ~/documents $ how many lines are in foo.txt
Model predicted "wc -l foo.txt" (Confidence 0.89)
Would you like to execute this?
"yes" to confirm. "e" for more explanation:
```

```
query {
  x: "how many lines are in foo.txt"
  y_type: "CommandSequence"
}
```

```
predictions {
  {
    y_str: "wc -l foo.txt"
    y_ast: ...
    confidence: 0.89
  }, ...
}
```

Specification of Programs

```
  - define_new: Object
    type: Program
    name: wc
    children:
       - name: l
       - name: w
       - name: c
 - define_new: Object
   type: Program
   name: pwd
```

Declaration of Examples

```
x:
  - get the line count of [-[FILENAME]-]
  - how many lines does [-[FILENAME]-] have?
  - count num of lines in [-[FILENAME]-]
y:
  - wc -l [-[FILENAME]-]
x:
  - where am I?
  - print my working directory
y: pwd
```

Figure 1: A high level diagram of the AInix Platform

3 Background and Motivation

3.1 Relevant Terms and Concepts

Broadly speaking, the main component of this project fits into the field of semantic parsing. Semantic parsing is a subfield of natural language processing in which the goal is to convert from natural language to a formal representation. The exact specification of this formal representation can be highly variable. For example λ-calculus (Zettlemoyer and Collins, 2005) or abstract meaning representations (Banarescu et al., 2013) can be used as generic representations of meaning. When the goal of the system is interacting with knowledge bases to answer user questions, SQL queries (Hemphill et al., 1990), Prolog (Zelle and Mooney, 1996), or more complex query graphs (Zhang et al., 2016) might be used. For other problems, one might use domain specific representations such as those for processing If-This-Then-That (IFTTT) scripts (Quirk et al., 2015) or languages for representing regular expressions (Ranta, 1998; Locascio et al., 2016).

A subfield of semantic parsing is natural language for code generation. Prior work in this includes generating Python (Yin and Neubig, 2017) or Java (Ling et al., 2016) code from natural language descriptions.

While λ-calculus or general purpose programming languages can represent complex semantics which involve composition or other complex relations, another side of semantic parsing focuses on intent classification and slot filling. In this, the user's utterance is first classified into one of many intents. The intent then might have slots for entities which are typically filled by selecting string spans directly in the user's input utterance (Liu and Lane, 2016). Such an approach is popular in commercially deployed dialog systems like Amazon Alexa (Kumar et al., 2017). Intent/slotfilling is relatively simple and can be both accurate and scalable with current techniques. Starting with a slot filling approach, others have explored how extra complexity and compositionality can be added where needed (Gupta et al., 2018).

3.2 Currently Available Platforms and Tools

3.2.1 Digital Assistant Platforms

Recent years have seen massive mainstream adaption of natural language interfaces and digital assistants. In a survey of 5000 US consumers conducted by Microsoft in February 2019, it was found that 69% of respondents have used a digital assistant. This is rapidly expanding as smart speaker ownership grew 22% in 2018 and is expected to continue to grow. Apple's Siri and Google Assistant lead digital assistant adoption with each having a 36% user share. Amazon's Alexa comes in third with 25% followed by Microsoft's Cortana with 19%. This leaves only 1% for other. (Olson and Kemery, 2019).

There are many assistants in this "other" category. For example, Mycroft[1] is an open-source and privacy-focused digital assistant which has been shipping smart speakers since having a successful Kickstarter in 2016. While Mycroft has shipped thousands of devices, the device's utility is limited compared to the market-leading devices due to the size of the ecosystem. While Mycroft has around 100 skills[2], Amazon Alexa has about 70,000 skills[3] which are created using Amazon's developer platform, Amazon Voice Services (AVS). Other smaller open source assistants and other commercial products such as Jibo or Bixby all struggle with small ecosystem size. Both subsection 3.3 and section 9 discuss ways AInix will attempt to avoid a similar fate.

3.2.2 Semantic Parsing Frameworks

If one is not looking for a complete digital assistant platform, and instead only an open source semantic parsing framework, there are many options. These include SEMPRE (Berant et al., 2013) and Cornell SPF (Kwiatkowski et al., 2013) which are based on non-neural models, and more recent frameworks like TranX (Yin and Neubig, 2018) and AllenNLP (Gardner et al., 2018) which are based on deep neural techniques. These libraries are all either fairly low-level or more research-focused. While there is a need for such frameworks, it is a different use case than what the AInix Project target at. For the thousands of developers using tools like Amazon Voice Services, Google Dialogflow, or IBM's wit.ai, they used these tools not because they were looking to perform "semantic parsing", they used them because they wanted an easy way to create a language interface which could be deployed across a large number devices. AInix hopes to work towards a similarly developer friendly and high-level platform, while remaining open-source and customizable.

3.2.3 Previous Natural Language to Shell Commands Work

Very relevant to the AInix Project is prior work creating the Tellina system and the NL2Bash corpus (Lin et al., 2018). In this dataset there 9,305 examples gathered from examples and questions on the web. These cover 102 different utilites. Because these came from questions on the web, many of these commands are very complex. Lin et al. present two different categories of models. One being a Seq2Seq model with a pointer-based copy mechanism, and another being a template-based approach (an introduction to Seq2Seq models is provided in subsection 7.1). In the template-based approach, first a seq2seq model is used to generate a template (program sketch), then a alignment model and heuristics are used to fill constants into the template. They found that the Seq2Seq with a copy mechanism performed the best and achieved an accuracy of about 31%. They released source code and a web demo for interacting

[1]https://mycroft.ai/

[2]https://github.com/MycroftAI/mycroft-skills

[3]https://www.theverge.com/2019/1/4/18168565/amazon-alexa-devices-how-many-sold-number-100-million-dave-limp

with the model, but do provide a built out way for developers to create systems on top of it, or a way for users to use Tellina in a shell session.

3.3 Why Unix Commands?

A question a reader might have is if the long term of goal the AInix Project is to create an open source platform for language interfaces, why start with Unix shell commands? Why not focus on creating an open platform for problems where language interfaces have already been proven useful and popular, like in a voice-based digital assistant for tasks like playing music, looking up quick facts, or ordering something online?

As discussed in subsubsection 3.2.1, an ecosystem of skills are critical to building out a platform that is useful and attractive to users. A platform based on shell commands offers several advantages when trying to build out such an ecosystem. These include:

1. Shell commands are a universal interface. Due to historical reasons discussed further in subsubsection 3.3.1, shell commands are exceedingly cross-platform and can be used with software written in nearly any language. A search on Github for "CLI" (command line interface) will reveal tens of thousands of results. Because of the massive number of problems which an existing shell utility can solve, having a platform that can map from language to shell commands simplifies the process of adding new skills to the platform.

2. Being useful to developers helps builds an ecosystem. Remembering Unix shell commands is a problem that many developers and technical users have. By first focusing on a platform that helps developers, AInix can work to engage this critical demographic and give developers a vested interest in contributing back and building on the platform.

3. Avoids competing with existing large players. Companies like Amazon and Google have already invested heavily in the building very capable assistants and semantic parsers for domains with general appeal. Open-source consumer-focused platforms like Mycroft must try to match the high expectations set by these companies' products, and when the capabilities fall short, users might be inclined to leave the platform. In contrast, by focusing on the very niche domain of core Unix utilities, AInix does not have to directly try and compete or meet these user expectations. Thus we can initially provide some benefit to users, and then gradually grow into more mature areas as the platform becomes more developed.

3.3.1 Brief History of the Unix Shell Commands

To understand the points above, it is useful to understand some of the history of the Unix operating system. Unix began in the early 1970s at Bell Labs (Ritchie and Thompson, 1974). Since then it has grown to be the basis for many modern operating systems like Linux and Mac OS.

Part of what made Unix successful was the philosophy of being very modular (Raymond, 2003). Unix introduced the concept of pipeing, which allows users in a command-line shell to make the output of one modular command be the input of another command. By combining utilities in this way, more complex goals can be achieved.

Before the rise of GUI's made computing more accessible, some early AI research in the 1980s worked to make it easier to use Unix commands, as it was seen as necessary to help more people use computers. Such systems include Berkeley the Unix Consultant (L. Graham and Wilensky, 1987), Unix Softbots (Etzioni et al., 1994), and USCSH (Matthews et al., 2000). Like many expert systems of the time, these systems all typically required fairly complex rules and knowledge representations, and did not make use of the powerful machine learning techniques available today.

While GUIs make knowing shell commands no longer required for computing, shell commands are still widely used among developers and technical computer users. A survey of ~90,000 developers conducted by the popular question-and-answer site Stack-Overflow[4], found that 36.6% of developers use Bash/Shell/Powershell scripting, making it the 6th most commonly used programming language. This makes Unix-shell commands an attractive choice for the meaning representation for a semantic parsing framework. In contrast, languages like Clojure that resemble the lisp-like notation used in many semantic parsing frameworks are used by less than 2% of developers. This creates an extra barrier for developers wanting to use such frameworks.

4 Platform Architecture

AInix is designed with modularity in mind. A high-level diagram of the architecture is provided in section 4.

The main component is the AInix Kernel. This can be thought of as a semantic parsing framework. It provides the functionality that given a string utterance and desired type, it will predict the most likely translation of the utterance to the desired type. It defines a flexible interface for compatible models. If one wishes to experiment with a new model, one can adhere to the interface, and the kernel will handle feeding the model training examples, evaluating the training process, and serving the trained model.

Users interact with the Kernel through "AInix-shells". An AInix-shell is intended to be able to take many forms, such as a text-based terminal, a voice-based interface, a robot processing user commands, or many others. In subsection 4.1, we describe aish, which is a command line-based AInix-shell. As long as the interface is honored, AInix-shells could potentially interact with other semantic frameworks besides the AInix Kernel.

A separate module is AInix Common. This provides generic functionality for defining and parsing grammars as described in section 6, and communicating between AInix-

[4]https://insights.stackoverflow.com/survey/2019

shells and the kernel. This is used within the AInix Kernel, but open to being used in other projects which might find things like parsing shell commands useful.

Each component attempts to follow sustainable software engineering practices with documentation and regression testing.

4.1 aish

aish is intended to be both a useful tool and a reference implementation of an AInix-shell on the platform. It acts like a standard command-line Unix-like shell. It follows one goal very closely - help a user when possible, but never get in the way of existing workflows. What this means is that a user can use normal commands just as if using Bash or zsh, thus continuing their normal workflow. However, the user can also intermix natural language to fill in potential gaps in their knowledge.

When the user provides a command, it is passed through an Execution Classifier. This determines whether it should be executed as a shell command, or be passed into the AInix Kernel for parsing. Currently, this classifier is very simple and considers only whether the first word of the input is on the user's path or not.

When receiving a prediction from the model, aish provides basic functionality to help explain predicted commands by displaying relevant sections of man pages.

The shell itself is built off the xonsh[5] shell, a feature-rich, Python-based command-line shell. This helps provide functionality like tab-completion and command history, which users expect in a command-line shell.

5 AInix Kernel Dataset

A custom dataset of natural language and UNIX command pairs was created. This dataset was designed to address some of the more unique challenges of the task and be flexible enough to adapt to the long term goals of the project. This dataset is referred to the AInix Kernel Dataset.

5.1 Key Features

5.1.1 Many-to-many

A challenge of converting from language to commands is that multiple commands can correctly complete a task. To handle this, all examples in the AInix Kernel Dataset are added as many-to-many examples. An XValue (Source Value) utterance is associated with YSet made up multiple YValues (Target Values), which are all valid translations of the XValue. All values in a YSet are provided in preference order, with the default weighting being that each successive YValue is half as preferable as the previous one.

[5]http://xonsh.org/

```yaml
defines:
  - define_new: example_set
    # All example_set definitions must include a root y_type. For Unix commands,
    # the root type is CommandSequence. However, this framework could also be
    # used with a root type of RegularExp, SQL, or something else more domain
    # specific. This allows AInix to be used in multitask settings.
    y_type: CommandSequence
    examples:
      - x:
          - how many lines are in [-[FILENAME]-]
          - Counts lines of [-[FILENAME]-]
          - line count of [-[FILENAME]-]
          - get line count of [-[FILENAME]-]
        y:
          - wc -l [-[FILENAME]-]
        risk: 0
      - x:
          - how many words are in [-[FILENAME]-]
          - get the word count of [-[FILENAME]-]
          - word count [-[FILENAME]-]
        y:
          - wc -w [-[FILENAME]-]
        risk: 0
      - x:
          - how many letters are in "[-[ENGWORD]-]"
        y:
          - echo "[-[ENGWORD]-]" | wc -c
        risk: 0
    # ...Examples ommitted for conciseness...
```

Figure 2: A example of declaration of examples in the AInix Kernel Dataset

```yaml
defines:
  - define_new: example_set
    y_type: CommandSequence
    examples:
      - x:
        - list files here with smallest file first
        - display files smallest to largest
        - list files small to big
        - show files. Sort by file size small first
        y:
        - ls -lSr
        - ls -Sr
      # ...Other examples ommitted for conciseness............................
```

Figure 3: This YSet includes more than one YValue. Both "ls -lSr" and "ls -Sr" are valid translations of all XValues. However, because the query the cares about file size, the translation which includes the "-l" flag is likely more preferable so that file sizes can be seen in the printed output.

A weighting is not implied among XValues in terms of weighting examples during training or evaluation. However, when possible, the dataset is organized where the first XValue in the XSet is most the most descriptive. This first value is intended to be most appropriate as a representative XValue when explaining to a user the meaning of the YSet. For example, the utterance "List files sorted by modified date newest to oldest" and the utterance "ls sorted by mod date" could both be correctly translated into "ls -t". However, the first NL utterance is more precise, and if the system recommends the user execute "ls -t", it would be better if it explained to the user what the command would do using the first description.

Typically XValues are natural language, but are not required to be. For example, the dataset also includes "hybrid-language" (also referred to as code-mixing) such as in the XValue "ls -l sorted by size" with corresponding YValue "ls -l -S". Here a mix of both natural language and shell commands are used in this input. We see supporting hybrid-language important in making a practical system that can fit into existing workflows. It allows users to use the conciseness of Bash for things they know, but use natural language to help fill in the gaps of forgotten syntax.

This combination built-in many-to-many relations, preference ordering, and hybrid-language is not common in other datasets, but is useful for building out a usable system in this domain.

5.1.2 Replacers

```
defines:
  - define_new: example_set
    y_type: CommandSequence
    examples:
      - x:
        - make a copy of [-[FILENAME]-] in parent dir
        - copy [-[FILENAME]-] to parent directory
        - copy [-[FILENAME]-] up one directory
        y:
        - cp [-[FILENAME]-] ..
      - x:
        - copy [-[1=DIRNAME]-] to [-[2=DIRNAME]-]
        - recursive copy [-[1=DIRNAME]-] to [-[2=DIRNAME]-]
        - cp for directories [-[1=DIRNAME]-] [-[2=DIRNAME]-]
        y:
        - cp -r [-[$1]-] [-[$2]-]
      - x:
        - copy everything in [-[1=ENGWORD]-] dir to [-[$1]-]copy
        y:
        - cp -r [-[$1]-]/* [-[$1]-]copy
      # ...Other examples ommitted for conciseness............................
```

Figure 4: A series of examples making use of replacers. In cases where more than one replacer of the same type is used, it is necessary to assign them to a variable to disambiguate.

Perhaps even more so than traditional machine translation datasets, never-previously-seen named entity values are very common in Natural Language to Unix command translation. These entities include values such as file names or directory names. In order to augment the dataset and reduce the chance of overfitting, the dataset is created making extensive use of "Replacers". Replacers are placed in an example string and are randomly replaced during training. Some examples definitions which use replacers is shown in Figure 4.

To ensure reasonable replacer values for file names and directory names, we scraped

~223,000 files ending in .sh from Github using Google BigQuery[6]. We then parsed these files for strings known to be directory names such as the arguments to invocations of the "cd", "mkdir", or "rmdir" command, and stings known to be file names such as in invocations of "cat", "touch", "rm" (without -r flag), "tar -xzf", or "convert". We collected the top 10,000 Dirnames and Filenames along with the number of occurrences.

The number of occurrences is used to do a weighted sampling when replacing. This means that certain common directory names like "./src", "../..", or "./configure" are replaced into the examples more frequently than directories that only occurred once in the scraped files such as user-specific files like "src/proto/math/math.proto", "tmp/midijs/channels.js", or "429+4005527.fits". Similarly filenames such as "setup.py" and "log.log" occur much more frequently than filenames like "src/types/RectF.cs" or "/root/ho/v.0.0.1/hardeningone/tmp/report.txt". Replacers using data scraped from actual user files help ensure that models are trained with a realistic distribution of entities, rather than just a few common names or metasyntactic names such as "foo.txt" or "filename1" which do not represent real use.

Replacers for other entities were added such as for the 1000 most common English words (excluding stop words), most common file extensions, most common UNIX usernames, and English characters.

Replacers serve to augment the dataset and reduce the risk of overfitting. Further discussion of the effects of replacers on modeling is available in subsection 8.4.

5.2 The Arche Release

The current dataset is intended to facilitate initial experimentation and focuses on a very narrow set of commands. If a system claims to support a large number of commands but then a majority of the time produces incorrect outputs, the system would be neither trustworthy nor useful. Instead, it is better to focus on a narrow scope until a high-level accuracy can be achieved before gradually expanding the scope.

Current the AInix Kernel Dataset includes 434 natural-language/hybrid-language XValues. These 434 utterances are across 153 unique YSets (i.e. examples with different semantic meaning), implying that that on average each YSet is associated with 2.8 different phrasings. Each YSet has on average 1.1 YValues. In total 15 Unix commands are covered.

For comparison, the Tellina NL2Bash dataset includes on average 1.23 (median 1) NL values per cmd value (roughly comparable to the average 2.8 XVals per YSet of the AInix Kernel Dataset) and 1.09 (median 1) cmd values per nl value. The combination of containing complex commands and usually only providing only one phrasing per command makes the NL2Bash dataset much more difficult to model, and harder to reach levels of accuracy needed to provide a usable system.

The AInix Kernel Dataset is intended to be a "living dataset," and will have continuous integration of pull request additions from the community as gaps are identified

[6]https://cloud.google.com/bigquery/public-data/

and we work to add new supported commands. Because this natural language to bash commands task requires domain expertise beyond that available on sources like Mechanical Turk, we believe establishing community involvement will be important to realistically collecting a large dataset.

In order to allow comparable results between others who may wish to experiment on the dataset, we will periodically create named releases which will be distributed along with git SHA's of the dataset. These releases will be independent of the releases of the AInix Kernel, aish, or other components of the platform. The first named release of the AInix Kernel Dataset will be named Arche (named after a tiny moon of Jupiter). Subsequent releases will proceed alphabetically.

For evaluation, we split the dataset into a 80:20 train:test split. We do not perform a separate "development" or "validation" split. This is for two reasons. The first is that, because of the very small size of the current dataset, further subdividing the training split would make meaningful training more difficult. The second is due to how the dataset is intended to be used. Unlike many machine learning datasets, where there is a collection phase to gather the data before progressing to modeling and evaluation phases, the AInix Kernel Dataset is intended to be gradually added to through pull requests from domain expert users, and be constantly experimented with. Because experiments were constantly being run during development, we do not feel like we could claim in good faith that there is an untouched portion of the dataset to serve as a clean test split. In order to reduce the risk that reported results have been overfitted to a single 20% split, reported results are averaged over five retrainings on five random 80:20 splits.

6 AInix Grammar Files

AInix creates a type system and parsing grammar for the formal output commands which can be used to add inductive bias to models, and to better evaluate different string commands with equivalent semantic meaning. In prior work, using the inductive bias from such a grammar has been shown to be useful for improving semantic parsing (Rabinovich et al., 2017; Krishnamurthy et al., 2017). We wish to employ these same grammar-based techniques to Unix-like shell commands.

We create a simple way of defining the type system in YAML files[7]. YAML is a language commonly used in expressing software configuration files, and is similar to XML or JSON. This method of defining grammars is intended to flexible enough to accommodate the challenges of parsing Unix-like commands, and be familiar to other developers wishing to extend the type system.

[7]https://yaml.org/

```yaml
defines:
  - define_new: object
    name: wc
    type: Program
    # We must provide type data so the default TypeParser for Programs knows
    # how to recognize this object in strings.
    type_data: {invoke_name: "wc"}
    children:
      - name: l
        arg_data: {short_name: "l", long_name: "lines"}
      - name: c
        # By default arguments are optional.
        # We can also explicitly specify this for clarity.
        required: False
        # The default ObjectParser for Program's knows how to parse
        # POSIX-compliant program argument conventions. To do this the parser
        # examines the arg_data that the user provides for the arguments.
        arg_data: {short_name: "c", long_name: "bytes"}
      - name: m
        arg_data: {short_name: "m", long_name: "chars"}
      - name: w
        arg_data: {short_name: "w", long_name: "words"}
      - name: files_list
        # The previous arguments had None type (were just flags)
        # However, this positional argument has a Type which needs to be parsed
        type: PathList
        arg_data: {position: 0, multiword_pos_arg: True}
```

Figure 5: A code snippet showing the specification of the "wc" command.

```yaml
defines:
  - define_new: type
    name: Number
  - define_new: object
    name: decimal_number
    type: Number
    children:
      - name: SignArg
        type: Sign
        required: False
      - name: BeforeDecimal
        type: IntBase
        required: True
      - name: AfterDecimal
        type: IntBase
        required: False
      - name: Exponent
        type: IntBase
        required: False
    preferred_object_parser:
      grammar: |
        SignArg? BeforeDecimal ("." AfterDecimal)? ("e" Exponent)?
  # Definition of Sign type and IntBase type omitted ..........................
```

Figure 6: A code snippet that shows the declaration of a new type with a single object. In the declaration shown in Figure 6 we used an existing default parser which was written in Python and is customized with arg_data. Here we use the ability to generate a parser using PEG-like grammar.

6.1 Challenges of Parsing Unix Commands

Compared to parsing Python, SQL, or another DSL (domain specific language), shell commands are significantly harder. The POSIX standard defines a grammar for syntax like pipes, file redirects, determining what utility to execute, and how quotes or special characters should be handled. However, the format of actual arguments to commands is not well standardized or consistent. The space-separated words of the shell input are passed into the utility, and the utility may have any interpretation on those strings it chooses. This has results in many issues:

1. Parsing can be very context sensitive. Depending on the arguments that come before or after it, an argument can be either valid or invalid. In case of the use of pipes or utilities like xargs, this context can span between utilities.

2. Many different strings need to have the same semantic representation. For example "head -n 4 –quiet foo", "head -qn 4 foo", "head -q -n 4 foo", "head –lines=4 -q foo", "head -q -n4 foo" all have the same representation. This is a relatively very simple example.

3. Different utilities use different string representations for semantically equivalent arguments. For example, different commands represent arguments such as times, file sizes, or numbers differently. In order to best gain from defining grammars through improved transfer learning, these different string grammars all need to be mapped into identical ASTs.

6.2 Terminology

We define our Abstract Syntax Trees (AST) grammar in terms of Types and Objects. Types define a set of related Objects. An Object has zero or more children. The term Arguments of an Object is used interchangeably with the "Children" of an Object. As an example in the UNIX command domain, one Type is named "Program" which has objects named "ls", "cat", "mkdir", etc.

Abstract Syntax Trees are rooted with ObjectChoiceNode's. An ObjectChoiceNode is associated with a a certain Type T. An ObjectChoiceNode is proceeded with either an ObjectNode with an Object O of Type T or a CopyNode. An ObjectNode has an ObjectChoiceNode for every child of the Object O. A CopyNode is terminal and points to a start index and end index defining a span in a reference string that can be parsed into T. Note that this is different from many similar techniques as entire subtrees can be copied, not just terminal/primitive nodes.

Each of these Objects has varying number of arguments. Arguments can be either required or optional. If an Argument is optional then it may have None type. An example of an optional Argument with None type is the "-l" flag of the utility "ls". The "-n" flag of the "head" utility is optional but has type "Number". The directory parameter of the "cd" utility is both required and non-None.

It should be noted that creating Optional arguments is syntactic sugar, and that AST nodes do not track a concept of "present" or "not present". In reality, for every optional argument, AInix automatically creates a new Type with two Objects - an "ArgPresent" and "ArgNotPresent" Object. The ArgPresent object has one required child of the type specified by the argument. This syntactic sugar makes it more natural for programmers to express grammars like that in UNIX commands with optional flags, while also keeping the AST's and modeling simpler.

6.3 Enabling Flexible Parsing

It was important to have a parsing scheme that can both handle the quirks of parsing UNIX-like complex grammars as well as provide the maximal possible levels of abstraction for developers wishing to add their own programs or domains to the platform. To do this, we use two kinds of parsers.

A TypeParser parses ObjectChoiceNodes. They are tasked with: given a string, output which Object implementation is represented by that string, and the span of string to pass forward to parsing the arguments of that string.

An ObjectParser parses ObjectNodess. They are tasked with: given a string, output which of the arguments are present, and if the argument is present (or required), which span of the input string should be passed onto parsing that argument.

In the cases of parsing a simple POSIX-compliant Program, one can determine all this information simply given a string. The TypeParser can look at the first word to determine which Program object to select. An ObjectParser for a Program can look for the appropriate dashes and flags to identify which args are present and which substrings correspond with each argument. However, many grammars are easier to define when the substring of the argument is determined by the parse results of lower parsers, as in the case of LL recursive descent parsers (Rosenkrantz and Edwin Stearns, 1970) that are commonly used to parse programming language source code. AInix supports LL parsing by allowing ObjectParsers and ObjectChoiceParsers to yield parsing into lower parsers, receiving back whether the parse succeeded and the rightmost reach of the parse span.

In addition, TypeUnparsers and ObjectUnparsers handle converting ObjectChoiceNodes and ObjectNodes back into strings. The results of the unparses are automatically tracked so we know the string associated with every subtree of the AST.

TypeParsers, ObjectParers, and their respective Unparsers can be registered as arbitrary Python functions. This provides the flexibility to parse complex rules such as POSIX/GNU-style command flags. In addition, AInix grammar markup files also allow for the inline definition of LL-style parsers using a notation that is similar to a subset of Parsing Expression Grammar (PEG) (Ford, 2004). Based on the PEG-like notation, AInix will automatically create the appreciate parsers and unparsers.

Assigning of default parsers and the ability to assigning Object and Argument attributes help abstract away most of the complexities of parsing from a developer wishing to do something like adding a new POSIX-compliant utility on the platform.

This is demonstrated in Figure 6. Objects of type Program have a default ObjectParser which handles parsing POSIX-compliant flags. The "invoke_name" property set on the Object lets the default TypeParser for the ProgramType chose the appropriate object. The properties on the arguments communicate information like the name of the program, the short or long name of the argument, or if the argument is a positional argument.

The parsers for specific Objects or Arguments may be overridden, partially addressing the issue discussed in subsection 6.1 (see subsection 6.4 for why this is not yet fully addressed).

This grammar specification method provides a combination of both high expressiveness and abstraction, greater than that most semantic parsing frameworks.

6.4 Limitations

While able to handle many utilities, this grammar is lacking in many ways and needs many features. These include:

1. Sets. Currently, there is no way to represent order invariance. For example, a combination of find "and" operations are order invariant, as are char groups in regular expressions. Additionally, there is currently no way to enforce uniqueness constraints.

2. Currently, CopyNodes cannot appear in the middle of an AST as they copy entire subtrees. This could be solved with variadic arguments, which would eliminate the need to express lists as linked-lists.

3. Subprocess calls are currently not supported.

4. Utilities like xargs which introduce new syntax into a command pose challenges. We intend to address this using "macros" losely inspired by lisp/Clojure macros. This will allow an Object to inject a macro into the parsing, which can define a special action classification choice.

5. Recursive overriding. Currently, a TypeParser or ObjectParser can override the next parser. However, in order to better address the issue with equivalent semantic types having different string parses, it would be better if overriding could be applied to stick recursively.

6. Type Unions. Types should be able to be defined dynamically based off a composition or blacklisting of objects. Without type unions there is potentially a lot of redundancy in objects.

7 Explored Models

For the primary task of the AInix Kernel - The conversion of an utterance to an AST rooted at the specified type - a variety of modeling techniques were used. These broadly fall into three categories: string machine translation methods (seq2seq), methods to better exploit the structure provided by the grammar, and nonparmetric methods relying on retrieving examples from the training set.

7.1 Seq2Seq

Similar to the exploration by Lin et al. (2018) on the NL2Bash dataset, we apply Seq2Seq models to the AInix Kernel Dataset. A Seq2Seq model, or a Sequence-to-Sequence model, is a technique originating machine translation to convert from a string in one language, to a string in another language. This technique involves first taking the input sequence and converting it to a vector representation. This is typically done with a deep neural network such as an LSTM recurrent neural network. Once the input sequence is embedded in a vector space, the embedding is used to condition the output of a decoding network which sequentially outputs the translated sequence. This model is trained end-to-end with backpropagation to maximize the likelihood the desired output sequence is produced (R. Medsker and C. Jain, 1999; Sutskever et al., 2014; Gu et al., 2016).

In our case, before using the Seq2Seq model, data is preprocessed by passing the YValue through parsing and unparsing in order to normalize parts like flag order. Both the XValue utterance and YValue string representation are tokenized at non-letter and whitespace boundaries, with tokens explicitly added to represent spaces. We sample each XValue in the dataset 35 times. During each sample replacer values might be different. In addition, its corresponding YValue string is selected proportionally to its preference weighting in the YSet.

We use a Seq2Seq model based on OpenNMT (Klein et al., 2017). The model starts with a 300-dimensional embedding filtered to words appearing at least 40 times. Because of the multiple samples XValues, a higher than typical filter threshold is needed. The embedding is followed by a two-layer LSTM encoder with 128 hidden units, and a two-layer LSTM decoder with 128 hidden with input-feeding and global attention (Luong et al., 2015). A 0.3 dropout exists within the encoder, between the encoder and decoder, and within the decoder. We found architecture features in the base OpenNMT model like input-feeding helped improved performance, and did not achieve as accurate results using some other seq2seq frameworks.

We train with a batch size of 64 for 7500 steps using the default OpenNMT SGD optimizer, halfing the learning rate after 4000 and 6000 steps. For evaluation, we decode using a beam size of 10. We then take the resulting string and parse it. We consider it correct if the parsed AST appears in the ground truth AstSet.

7.2 Grammar Guided Methods

In addition, we explore using the AST grammar as part of modeling similar to Rabinovich et al. and Yin and Neubig.

In this case, one can think of every Type declaration defining a classification task over all of the available object implementations. Starting at the root ObjectChoice node a classification is performed on the root type. The decoder then proceeds recursively down the tree depth-first and left-to-right.

Similar to the seq2seq model, a 2 layer biLSTM is used to encode the input utterance. Then a LSTM decoder is used.

We learn an embedding for every type and every object. When encountering an ObjectChoiceNode, we enter an RNN cell. This RNN cell is conditioned on both the Type it is predicting and which Object it is a child of in order better inform the vector transformation the cell applies. More specifically, the following operation is applied:

Let t be the embedding of the type we are predicting.

Let p be the embedding of the parent object. A special learned vector is used if we are at the root and have no parent.

Let h be the hidden and cell state from the previously predicted node. Because we predict depth-first, this might not necessarily be a direct sibling to this ObjectChoiceNode. If we are at the root node, the hidden state comes from the encoder state.

$$s = f_{LSTM}(dropout([t : p]), h)$$

Where ":" is vector concatenation. The vector s is then used as the query to global dot-product attention. We do not learn separate query and context vectors in the attention.

The RNN output vector after the attention goes through two hidden feed-forward layers and a sigmoid to predict the probability that copy is chosen. If copy is chosen, the vector goes through two separate learned linear project which are doted with the encoded tokens to determine the start and end of the copy span. If a copy is not chosen, we must choose what object implementation to select. We do this with a linear project followed by doting with an embedding of every valid implementation. Because of the many-to-many nature of the dataset, multiple implementations might be valid. Therefore, instead of cross entropy loss, we use binary cross entropy for each of the implementations of the Type.

7.3 Retrieval Based Methods

In addition, non-parametric methods are explored. Prior work has found retrieval methods to be effective when predicting on source code (Hayati et al., 2018). Most prior work methods used a distance metric of the nearest neighbor derived from string edit distance or other metrics which lack a clear tie to semantics. We explore a nearest-neighbor based technique using deep contextual embedders. This embeds the training examples into a vector space, and then uses those embeddings when encountering an unseen example.

7.3.1 UNIX-focused contextual embedding

In order to help with this task, we created CookieMonster, a contextual embedder focused on the needs of the project.

7.3.2 Background

Contextual embedders focus on creating a dense embedding for every token in an input string which embeds not only the meaning of the word, but also the meaning of the word in that context. Contextual embedders such as ELMo (Peters et al., 2018) and BERT (Devlin et al., 2018) have shown this to be effective in improving downstream tasks. These contextual embedders use language modeling tasks to gain a complex understanding of words in context. A language modeling task involves trying to learn probability distributions of strings in a language. For example, a language model might learn that if a sentence starts with "I went to the", it is more likely to end with "store" then it is to end with "watermelon". By learning parameters which do well on language modeling large amounts of data, contextual embedders can learn a complex understanding of words in context.

In the case of BERT, a model is trained on two tasks. The language modeling task is phrased as - given a sentence with 30% of tokens either masked out or swapped with a different token, create an embedding for the masked token which can be used to predict the missing tokens. The second task is to predict whether two sentences concatenated together are actually sequential or not. See Devlin et al. (2018) for a more complete description of the task.

Existing contextual embedders like BERT have some issues which do not make them ideal for the AInix Platform. First, these models are not specifically tuned for text related to Unix commands and code. Second, these models are very larger in both parameter count and computational cost, which makes it difficult to run at interactive speeds on limited hardware.

7.3.3 Collecting a code-focused dataset

BERT uses a combination of the BooksCorpus (Zhu et al., 2015), which is a collection of 11,038 books from the web, and English Wikipedia for a total dataset with around 3.3 billion words.

To collect more Unix-command focused commands, we primarily use posts from different Q&A sites on StackExchange (the makers of StackOverflow). In particular, we download archives provided by Archive.org[8] for unix.stackexchange.com, serverfault.com, askubuntu.com, and superuser.com. Posts include both questions, and an-

[8]https://archive.org/details/stackexchange

swers, but not comments. We filter out any posts with a score less than -1, contain a sentence with less than 4 words or 16 characters, contain a sentence with longer than 320 characters, a code block with more than 160 characters, or less than two sentences. This was filtering was done to prevent including sentences in the modeling task without much context, or which were so long that they would complicate batching. This results in about 1.2 million posts, 6.1 million sentences, and 46 million words.

To further augment the data collected from StackExchange, we drew inspiration from the WebText dataset from GPT-2 (Radford et al., 2019) and scraped the social media platform Reddit for relevant articles. WebText was created by taking all outbound links from all subreddits with at least 3 upvotes over the span of 1 year. Instead, we only take links from 186 subreddits but widen the time range to take posts from January 2014 to February 2019. The selection of these subreddits was biased towards things with some relevance to Unix commands, but also included some more general interest topics to add some amount of diversity to the dataset. We only include articles that meet a certain upvote threshold. This threshold was set on a per-subreddit basis in a way that was roughly inversely proportional to the size of the community and directly proportional to our subjective and imprecise judgment of its relevance to the task. For example, links from the r/bash and r/commandline were taken with a threshold of 0 upvotes, while links from r/Aquariums and r/radiohead were taken only if having at least 200 upvotes.

This resulted in about 50 thousand articles. When combined with the StackExchange posts, it resulted in a total of about 130 million words and 700MB of text. Thus the dataset is approximately 4% the size of the BERT dataset and 2% of the GPT-2 dataset, but is biased towards text relevant to the task and remains manageable even on comparably modest hardware. While much smaller than these massive datasets, it is about 26% larger than the WikiText-103 dataset (Merity et al., 2016).

7.3.4 CookieMonster Model

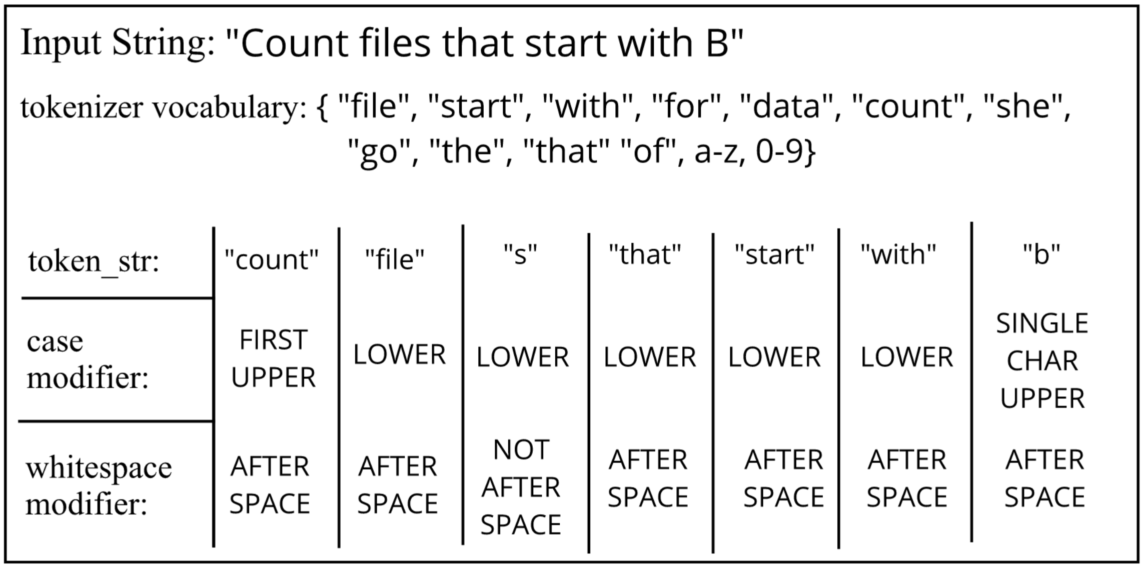
We train CookieMonster on the same task as BERT, but using our code-focused dataset.

7.3.4.1 CookieMonster Tokenization Scheme

We first learn 5000 uncased word pieces using Google's sentencepiece library[9] (Wu et al., 2016). This is an unsupervised way of finding the approximate best 5000 tokens which will tokenize the dataset in the least number of tokens. This allows for more efficient learning of out of vocabulary words such as "greped" which might be a combination of "grep" and "ed".

In addition, we use a novel tokenization scheme which we refer to as a ModifiedWordpieceTokenizer. This separates out the embedding of knowledge about words, whitespace, and casing.

[9]https://github.com/google/sentencepiece


Input String: "Count files that start with B"

tokenizer vocabulary: { "file", "start", "with", "for", "data", "count", "she", "go", "the", "that" "of", a-z, 0-9}

token_str:	"count"	"file"	"s"	"that"	"start"	"with"	"b"
case modifier:	FIRST UPPER	LOWER	LOWER	LOWER	LOWER	LOWER	SINGLE CHAR UPPER
whitespace modifier:	AFTER SPACE	AFTER SPACE	NOT AFTER SPACE	AFTER SPACE	AFTER SPACE	AFTER SPACE	AFTER SPACE

Figure 7: An example of a string tokenized with the ModifiedWordPiece tokenizer.

Normally, if one desires to capture case information about WordPiece tokens, one would learn two separate word embeddings. For example, the strings "use", "Use", and "USE" would all need different embeddings, even though their meaning is nearly identical. One might just appear in the middle of the sentence, one might appear at the beginning of the sentence, and one might appear in certain contexts for emphasis. It seems prudent instead to learn the casing information as a modifier to the meaning of the word instead.

Similarly, in a standard WordPiece tokenizer the token string "Data" when as a single word (whitespace on either side) and the token "Data" when appearing as part of the cammelCase variable "clientDataManager" would have different embeddings. This is because whitespace is learned separately in a standard WordPiece tokenizer, and the actual token might be "_Data" where the underscore represents a whitespace. By learning whitespace information separately, we can avoid this redundancy.

Like in a WordPiece tokenizer, a string is tokenized greedily based off the vocabulary. However, in addition to the vocab string of each token, we also store the case modifier and whitespace modifier. A case modifier is either LOWER, FIRST_CAP (in which only the first letter is capitalized), ALL_UPPER, CASELESS (which is assigned tokens made up of only symbols such as "(" or ">"), and SINGLE_CHAR_UPPER (which is a special case of when the token is only a single character and capitalized). A whitespace modifier is either AFTER_SPACE_OR_SOS or NOT_AFTER_SPACE. Like standard WordPiece tokenizers, this tokenization scheme is for the most part lossless, meaning the original string can be recovered. An exception to this is instances where there is more than one space between tokens.

If we wish to embed our tokens in N dimensional space, a different is N dimensional

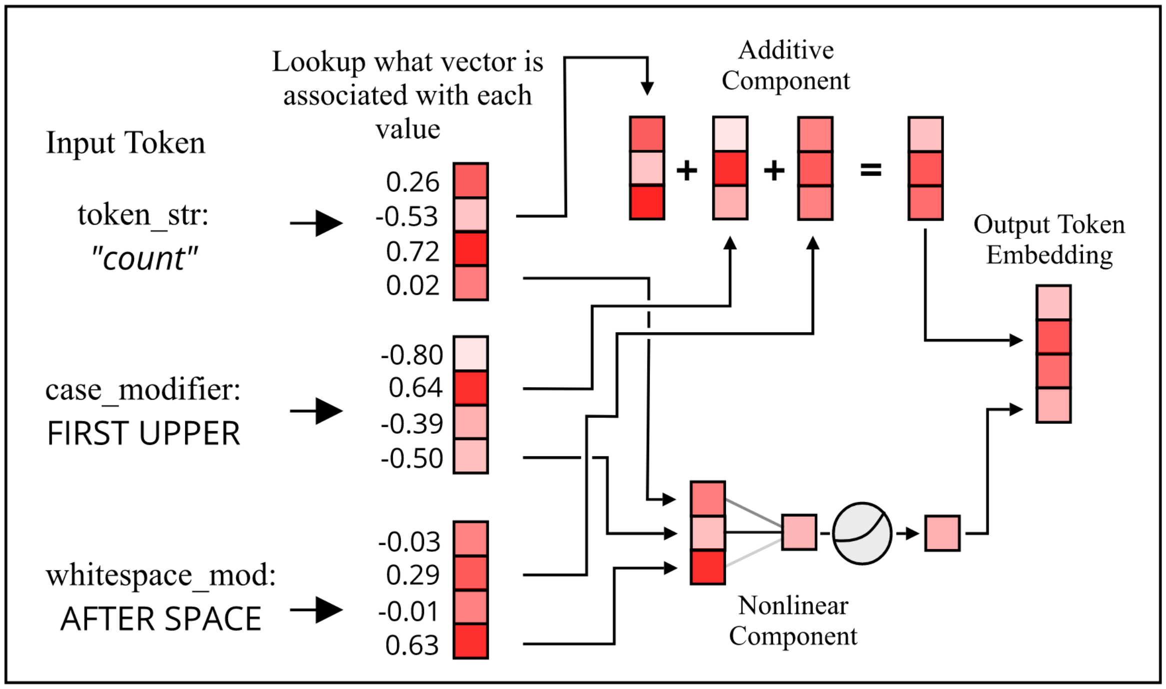


Figure 8: A visualization of embedding a ModifiedWordPiece token. In this toy example we use $N = 4$ and $\alpha = \frac{3}{4}$

embedding is learned for each vocab string and each modifier. To get the embedding for single token the following is performed. The first α $(0 \leq \alpha \leq 1)$ fraction of the N dimensions of the string, case modifier, and whitespace modifier embeddings are added elementwise to form vector $\mathbf{a}$. The remaining $(1-\alpha)N$ units of each vector are concatenated into a vector of length $3(1-\alpha)N$ which is passed through a learned linear projection back to a $(1-\alpha)N$ vector followed by a GELU (Hendrycks and Gimpel, 2016) non-linearity. The vector after this non-linearity is then concatenated with $\mathbf{a}$ to arrive back a vector of length N. For CookieMonster we use $\alpha = \frac{3}{4}$.

The reason for doing an elementwise addition on a fraction of the embeddings and a learned combination of the other fraction of the embeddings is based off on the assumption that there is likely not an interdependence between most of the knowledge captured in the vectorspaces of the strings and the modifiers. An alternative to this scheme might have been to add a one-hot unit to the string embedding which encodes the modifiers, then pass that $(N + 5 + 2)$ length vector through a projection back to N a length vector. However, this would have required $((N+7)N)$ learned parameters, which intuitively seemed excessive to capture the interrelation between the modifiers. In addition if one wished to have a nonlinearity on this vector, such a scheme would clip the entire vectorspace of the token embedding, rather than just a fraction of it.

The CookieMonster tokenizer also includes a modification where the internal tokens of long filenames are merged into one token. Hand-engineered heuristics are added to detect potential filenames in a sentence based off features such as the inclusion of forward slashes and a single dot. If a whitespace-separated word is more than three tokens long, and is recognized as likely a file, all but the first and last token is replaced with a special "<mergetok>" token.

When left unmerged, a typical file name might consist of five or more tokens. Long files names consisting of few English words might get tokenized into twenty or more tokens. The content of such paths typically have relatively little semantic value, but add substantial potential noise to the embedding model and increase computation costs. By merging the middle of such words, we reduce this concern while still retaining information such as file extension, which does provide critical semantic information such as when trying to differentiate between the proper utilities in the utterances "extract ~/projects/sNG/oldsrc/fa3.tar into ./savesrc/" and "extract ~/projects/sNG/oldsrc/fa3.7z into ./savesrc/" (Note how without merging, such a file might get tokenized into perhaps around 17 mostly meaningless tokens "~ / project s / s N G / old src / f a 3 . tar" → "~ <mergetok> tar"). In addition to helping reduce noise in encoders, merging tokens is useful for allowing average-polling based approaches of getting fixed length descriptions of sentences, as employed in subsubsection 7.3.5. This comes at the cost of reduced expressiveness of the models in potential cases where the middle tokens are semantically relevant, and the extra complexity of ensuring metadata is maintained to facilitate proper copying.

7.3.4.2 CookieMonster Architecture

The CookieMonster model is currently very small. It starts with a 192-dim embedding of tokens. Unlike in the BERT model, we do not add an embedding to distinguish between the first and second sentence. Then a 1d convolution is applied with a kernel of 3 and separated into 4 groups (Kaiser et al., 2017). This is followed by a two-layer biLSTM. Finally, there is a transformer (Vaswani et al., 2017) with 12 heads (where each head is 16 units). During prediction of the language modeling task, a linear projection to the vocab size for masked words is added. The next sentence task is predicted by average pooling of tokens, and taking the polled value through feedforward classifier with one hidden layer of size 96 and GELU activation.

This results in a model with about 2.4M parameters. For comparison, the BERT model has about 108M parameters.

We train this model for about three epochs on the code-focused dataset. Note that the model had not yet converged, and with additional compute time performance could likely improve.

7.3.5 Nearest Neighbor Method

We use CookieMonster to explore a comparatively simple nearest neighbor method. During training, we first embed each training example using CookieMonster. We condense the contextual embedding a single vector by average pooling across all tokens. During the average pooling, we do not include stop words and weight down tokens inversely proportional to how long of a whitespace separated word they are apart of. This means that if the word "can't" is tokenized as "can ' t" each of those three tokens would together count as 1 token when average pooling. In order to cover the variance in embeddings for different instantiations of replacers, we sample 35 times and take the average.

During inference, we first embed the input utterance. We then look for the training example with the highest cosine similarity to the input utterance. The AST associated with this utterance serves as a template for the value to return.

In order to accommodate CopyNodes in the returned utterance template, during training we also store the average embeddings of the start and end of the span for each CopyNode. To improve robustness we also include the average for embedding for the token to the left and right of the start and end index as well. This forms the weighting for two different 1D convolution kernels. For the start-of-span kernel, we scale the averaged tensors by [0.25, 0.7, 0.05] and by [0.05, 0.7, 0.25] for the end-of-span kernel. This means that the token associated with the start or end span takes 70% of the kernel activation. We convolve this kernel over the new utterance, with the highest kernel activations becoming the predicted start and end span of the CopyNodes in the returned utterance.

A number of hand engineered heuristics are added to help with the process. Extra features were concatenated to tokens indicating whether they occur at the start or end of a whitespace boundary and whether they are part of span which has already been selected as the start, end, or inner span or a CopyNode earlier in the AST. Each these features are scaled to be about 0.1 of the mean L1-norm of the 192-dimensional embedding vector across all tokens. These hand engineered features help prevent the model from selecting the same span for multiple arguments.

8 Results and Discussion

Model	Accuracy
Seq2Seq; GloVe	52.4 ± 3.8
Seq2Seq; GloVe; No Beam Search	49.3 ± 4.4
Abstract Syntax Tree Network (ASN); GloVe	53.3 ± 2.2
Nearest Neighbor; CookieMonster	45.6 ± 4.1
Nearest Neighbor; base-BERT	36.0 ± 1.87

Table 1: Experiment results on the AInix Kernel Dataset. Experiments are over five random restarts using different random train/test splits. Results reported as mean accuracy on the held out test set and a 90% confidence interval on the mean.

Primary results are listed in Table 1. The following sections discuss the results of each model.

8.1 Seq2Seq Results

Similar to prior work, we find adapting NMT techniques and using a Seq2Seq model can be surprisingly effective. This serves as a baseline for the other approaches.

8.2 Grammar Guided Model Results

With our current implementation and dataset, we do not find that using Abstract Syntax Tree networks (ASN) significantly improve accuracy over a seq2seq model, but might serve some benefit helping reduce the variance of different training.

With a more complete implementation, these results might change. We do not implement beam search in the ASN. In addition, while dataset was created to only mostly only include commands within the grammar, due to restrictions in subsection 6.4, a small number (~1%) of dataset examples do not include proper copying, and could not be properly modeled by the ASN even with perfect accuracy. We find that it is likely that beam search offers a small improvement for a seq2seq model, so it is possible that if the ASN implementation was enhanced with beam search and a more expressive grammar, it might show a statistically significant improvement over Seq2Seq. Additionally, other ASN variations such as parent feeding or self-attention could be explored. However, with currently available evidence, we find it unlikely the ASN framework will offer a large improvement in accuracy (<5%). As the dataset grows in complexity and there is greater opportunities to gain from transfer learning between tasks sharing common types, it is possible this gap might widen, but this is not guaranteed. Any accuracy gains from ASN comes at the significant costs of requiring a grammar for all types one wishes to parse, increased model complexity, and increased training times (due to difficulties batching the decoding process).

8.3 Nearest Neighbor Model Results

We find the nearest neighbor model to be not as accurate as either the seq2seq model ($p = 0.04$) or the ASN ($p = 0.02$). However, considering the simplicity of the model, we are very encouraged by the level of accuracy that is achieved. Even if the nearest neighbor approach is not as accurate as the recurrent neural network approaches, significant other benefits including:

1. Faster training. The nearest neighbor model trains at least two orders of magnitude faster.

2. More explainable. The system can always point to a exact example with a known translation to explain what a command does. When executing commands, it is important to be able to present such explanations for confirmation from the user.

3. More predictable and frictionless continuous learning. It is very easy to add a new training utterance and then have confidence that if that utterance or a

very similar phrasing is encountered again, the model will be able to output the corresponding translation. In contrast, when training the RNN-based models, one must do significantly more retraining to avoid catastrophic forgetting and do not have as many guarantees that the new example will be incorporated into the model. It is important that users who find an utterance that is parsed incorrectly are able to easily contribute new examples, and then as immediately as possible see a result. This can help encourage further contributions to the dataset and makes the platform more useful.

Because these benefits, we see non-parametric nearest-neighbor-based techniques as most promising to meet the goals of the project. In section 9 we discuss some of the planned improvements to the nearest neighbor model to hopefully be able to closer meet or exceed the accuracy of the RNN-based models.

We also explore using BERT to embed utterances rather than CookieMonster. We use the pretrained base cased BERT model[10]. Because a "<mergetok>" is not in the vocabulary of the BERT tokenizer, we instead use the "##unk" token in places where long file names would be merged had one been using the CookieMonster tokenizer. Without doing any token merging, the BERT model accuracy is reduced. In addition we find using the "[CLS]" embedding as the fixed-length summary of the utterance does not perform as well as average polling. The result present in Table 1 uses both merging long paths as "##unk" and an identical average pooling scheme as when using CookieMonster.

We find strong evidence ($p < 0.01$) that using the CookieMonster encoder trained on code-focused data can outperform using the base-BERT model which was trained on Wikipedia + BooksCorpus. This is despite the fact that the BERT model is nearly 45x larger in parameter count and trained using approximately 800x more compute (a very rough estimate). We have not yet explored finetuneing BERT on the code-focus data, because even if better accuracy is achieved, using such a large model at an acceptable level of latency would require compute, memory, and storage requirements far greater than the minimum requirements the AInix Kernel is designed to support. We view training CookieMonster for more than 3 epochs (with perhaps slightly increased model capacity) as a more promising path to improve the contextual embedder in the short-term future.

8.4 Effects of Dataset Decisions

We explore the effects the replacers in the dataset with results shown in Table 2. We find when taking only a single replacement (were each example only has each has one randomly selected named entity), it results a significant drop in accuracy of the Seq2Seq model of around 22% percentage points. In addition, taking only one replacement sample rather than averaging over multiple replacements likely causes a small reduction in the accuracy of the nearest neighbor model.

[10]https://github.com/huggingface/pytorch-pretrained-BERT

Model	Accuracy
Seq2Seq	52.4 ± 3.8
Seq2Seq - Single Replacement	29.3 ± 6.5
Nearest Neighbor	45.6 ± 4.1
Nearest Neighbor - Single Replacement	43.0 ± 4.4

Table 2: Comparing models with and without replacements

This helps provide evidence there is a benefit to including replacers as part of the dataset collection process for this domain.

9 Future Work

Work on the AInix Project has really only just begun. Before inviting others to use AInix, we plan to do several things:

- **Nearest Neighbor Compositionality**: Experiment with extracting phrases on the AST similar to the ReCode system of Hayati et al. (2018). However, instead of edit distance, we plan to continue to use the contextual embedding.
- **Improve Parsing**: Fix at least some of the issues in subsection 6.4 (most pressing are those related to variadic arguments and CopyNodes)
- **Easier continuous learning**: Improve interface for adding new training examples, and be able to add new types or objects without retraining from scratch.

This will likely be sufficient for a first public release. Looking in the beyond that, there are several improvements to make.

On the AInix Kernel Dataset improvements include:

- **World state mocking**: Models should be able to check facts about the world. This can include things like whether a string is an existing file on the system or not, or other facts which require querying into a knowledge base. The dataset will support the ability to mock world state.
- **Risk labels**: every training example should be associated with a value signifying the risk that executing that command could cause inconvenience or harm to the user if that was not actually the intended translation. For example "wc -l foo.txt" has low risk, "tail foo.txt » bar.txt" has moderate risk, and "rm -r foo" has high risk. Being able to model potential risk is an essential part of building a useful and trustworthy system. A combination of well-calibrated model translation confidence values and predicted risk values can be useful information in deciding the amount confirmation to solicit from the user before executing. Risk is often word state dependent. For example "cp foo.txt bar.txt" is pretty

harmless, unless bar.txt already exists in the file system. There needs to be ways for accommodating this.

- **Partial YValues**: In some utterances like "create a copy of foo.txt", there is an ambiguous argument. Here the translation might be "cp foo.txt [-[?FILE 'what would name the new file?']-]". This could then allow an interactive dialog with the user.

AInix-shell improvements include:

- **aish improvements**: Broader syntax support and bug fixes are needed for aish to become viable for day-to-day use without interpreting existing workflows.
- **Alternative shells**: Demonstrate that the platform can be used for general appeal tasks like checking the weather or playing music. Demonstrate that other kinds of shells are possible like a primitive voice shell.
- **Shell agnostic rich output**: Develop a datacentric communication spec which can allow answering more complex queries (like those that require showing an image, a map, or a webpage), and be interoperable between different kinds of shells.

Work on making this roadmap more cohesive still needed as well. Such goals are only achievable by building an open source community for the platform. However, we are optimistic that there exists enough need, that this will be possible.

10 Conclusion

In this report we presented the AInix Project. We do not claim to offer a new state of the art model for semantic parsing or program synthesis. However, AInix does provide the architecture and framework for a scalable, developer friendly, and open language understanding platform. This includes a new dataset specifically for practical use, a simple way of defining grammars that keeps abstraction in mind, and a demonstration of techniques like a slot-filled nearest neighbor with code-focused contextual embeddings which shows promise for creating models that are predictable and easily extendable. We hope to continue to build on this work to achieve the vision of a free and open platform for AI-assisted computing. You can follow the latest status of the project at AInix.org.

11 Acknowledgements

I would like to thank Dr. Ray Mooney for supervising this thesis, for providing plentiful guidance and advice, and for helping introduce me to research. I would also like to Dr. Greg Durrett for providing his advice and helpful discussions. Finally, I would like to thank my friends and family for providing support throughout.

References

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., and Schneider, N. (2013). Abstract meaning representation for sembanking. In *LAW@ACL*.

Berant, J., Chou, A., Frostig, R., and Liang, P. (2013). Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Etzioni, O., Lesh, N., and Segal, R. (1994). Building softbots for unix (preliminary report).

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N., Peters, M., Schmitz, M., and Zettlemoyer, L. (2018). Allennlp: A deep semantic natural language processing platform.

Gu, J., Lu, Z., Li, H., and Li, V. (2016). Incorporating copying mechanism in sequence-to-sequence learning. pages 1631–1640.

Gupta, S., Shah, R., Mohit, M., Kumar, A., and Lewis, M. (2018). Semantic parsing for task oriented dialog using hierarchical representations. *CoRR*, abs/1810.07942.

Hayati, S. A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., and Neubig, G. (2018). Retrieval-based neural code generation.

Hemphill, C. T., Godfrey, J. J., and Doddington, G. R. (1990). The atis spoken language systems pilot corpus. In *HLT*.

Hendrycks, D. and Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Kaiser, L., Gomez, A. N., and Chollet, F. (2017). Depthwise separable convolutions for neural machine translation.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. M. (2017). OpenNMT: Open-source toolkit for neural machine translation. In *Proc. ACL*.

Krishnamurthy, J., Dasigi, P., and Gardner, M. (2017). Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Kumar, A., Gupta, A., Chan, J., Tucker, S., Hoffmeister, B., and Dreyer, M. (2017). Just ASK: building an architecture for extensible self-service spoken language understanding. *CoRR*, abs/1711.00549.

Kwiatkowski, T., Choi, E., Artzi, Y., and Zettlemoyer, L. S. (2013). Scaling semantic parsers with on-the-fly ontology matching. In *EMNLP*.

L. Graham, S. and Wilensky, R. (1987). The berkeley unix consultant project. page 12.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. (2018). Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.*

Ling, W., Grefenstette, E., Hermann, K. M., Kociský, T., Senior, A. W., Wang, F., and Blunsom, P. (2016). Latent predictor networks for code generation. *CoRR*, abs/1603.06744.

Liu, B. and Lane, I. (2016). Attention-based recurrent neural network models for joint intent detection and slot filling. *CoRR*, abs/1609.01454.

Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., and Barzilay, R. (2016). Neural generation of regular expressions from natural language with minimal domain knowledge. *CoRR*, abs/1608.03000.

Luong, T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.*

Matthews, M., Pharr, W., Biswas, G., and Neelakandan, H. (2000). Uscsh: An active intelligent assistance system. *Artificial Intelligence Review*, 14(1):121–141.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *CoRR*, abs/1609.07843.

Olson, C. and Kemery, K. (2019). 2019 voice report: Consumer adoption of voice technology and digital assistants.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *Proc. of NAACL.*

Quirk, C., Mooney, R., and Galley, M. (2015). Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 878–888, Beijing, China.

R. Medsker, L. and C. Jain, L. (1999). Recurrent neural networks: Design and applications.

Rabinovich, M., Stern, M., and Klein, D. (2017). Abstract syntax networks for code generation and semantic parsing. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

Ranta, A. (1998). A multilingual natural-language interface to regular expressions. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 79–90.

Raymond, E. S. (2003). *The Art of UNIX Programming*. Pearson Education.

Ritchie, D. M. and Thompson, K. (1974). The unix time-sharing system. *Commun. ACM*, 17(7):365–375.

Rosenkrantz, D. and Edwin Stearns, R. (1970). Properties of deterministic top-down grammars. *Information and Control*, 17:226–256.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation.

Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696.

Yin, P. and Neubig, G. (2018). TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP) Demo Track*.

Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *AAAI/IAAI, Vol. 2*.

Zettlemoyer, L. S. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*.

Zhang, Y., Liu, K., He, S., Ji, G., Liu, Z., Wu, H., and Zhao, J. (2016). Question answering over knowledge base with neural attention combining global knowledge information. *CoRR*, abs/1606.00979.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *2015 IEEE International Conference on Computer Vision (ICCV)*.