New Resources and Ideas for Semantic Parser Induction

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Philosophie (Dr. phil.) genehmigte Abhandlung

Vorgelegt von

Kyle Richardson

aus Nashua, New Hampshire (USA)

Hauptberichter:	Prof. Dr. Jonas Kuhn
Mitberichter:	Dr. Jonathan Berant
Tag der mündlichen Prüfung:	11.10.2018

Institut für Maschinelle Sprachverarbeitung
der Universität Stuttgart

2018

Erklärung

Hiermit erkläre ich, dass ich, unter Verwendung der im Literaturverzeichnis aufgeführten Quellen und unter fachlicher Betreuung, diese Dissertation selbständig verfasst habe.

(Kyle Richardson)

Contents

Abstract

In this thesis, we investigate the general topic of computational natural language understanding (NLU), which has as its goal the development of algorithms and other computational methods that support reasoning about natural language by the computer. Under the classical approach, NLU models work similar to computer compilers (Aho et al., 1986), and include as a central component a semantic parser that translates natural language input (i.e., the compiler's high-level language) to lower-level formal languages that facilitate program execution and exact reasoning. Given the difficulty of building natural language compilers by hand, recent work has centered around *semantic parser induction*, or on using machine learning to learn semantic parsers and semantic representations from parallel data consisting of example text-meaning pairs (Mooney, 2007a).

One inherent difficulty in this data-driven approach is finding the parallel data needed to train the target semantic parsing models, given that such data does not occur naturally "in the wild" (Halevy et al., 2009). Even when data is available, the amount of domain- and language-specific data and the nature of the available annotations might be insufficient for robust machine learning and capturing the full range of NLU phenomena. Given these underlying resource issues, the semantic parsing field is in constant need of new resources and datasets, as well as novel learning techniques and task evaluations that make models more robust and adaptable to the many applications that require reliable semantic parsing.

To address the main resource problem involving finding parallel data, we investigate the idea of using source code libraries, or collections of code and text documentation, as a parallel corpus for semantic parser development and introduce 45 new datasets in this domain and a new and challenging text-to-code translation task. As a way of addressing the lack of domain- and language-specific parallel data, we then use these and other benchmark datasets to investigate training semantic parsers on multiple datasets, which helps semantic parsers to generalize across different domains and languages and solve new tasks such as *polyglot decoding* and *zero-shot translation* (i.e., translating over and between multiple natural and formal languages and unobserved language pairs). Finally, to address the issue

of insufficient annotations, we introduce a new learning framework called *learning from entailment* that uses entailment information (i.e., high-level inferences about whether the meaning of one sentence follows from another) as a weak learning signal to train semantic parsers to reason about the holes in their analysis and learn improved semantic representations.

Taken together, this thesis contributes a wide range of new techniques and technical solutions to help build semantic parsing models with minimal amounts of training supervision and manual engineering effort, hence avoiding the resource issues described at the onset. We also introduce a diverse set of new NLU tasks for evaluating semantic parsing models, which we believe help to extend the scope and real world applicability of semantic parsing and computational NLU.

Deutsche Zusammenfassung

Grundgegenstand dieser Arbeit ist das Problem des Sprachverstehens (Natural Language Understanding, im folgenden NLU). Das Ziel dieses Teilbereichs der maschinellen Sprachverarbeitung ist die Erforschung und Entwicklung von Algorithmen zur Schlussfolgerung über natürliche Sprache durch Computer. Der klassische Ansatz ähnelt methodisch dem des Compilerbaus (Aho et al., 1986). Die zentrale Komponente ist dabei ein semantischer Parser, der die natürlichsprachliche Eingabe (beim Compiler also die Eingabe in einer höheren Programmiersprache) in eine formale Sprache auf niederer Ebene übersetzt, um die Ausführung und Interpretation des Programms zu ermöglichen. Da Compiler für natürliche Sprachen von Hand schwer zu bauen sind, konzentrieren sich neuere Arbeiten auf die Induktion semantischer Parser oder auf das maschinelle Lernen semantischer Parser und semantischer Repräsentationen aus parallelen Daten bestehend aus Paaren von Text und Bedeutung (Mooney, 2007a).

Eine Schwierigkeit für diesen datengetriebenen Ansatz liegt in der Beschaffung der parallelen Daten; diese kommen nicht "in der freien Natur" vor (Halevy et al., 2009). Selbst wenn Daten verfügbar sind, ist die Verfügbare Menge an sprach- und domänenspezifischen Daten möglicherweise nicht ausreichend, um verlässliche Modelle zu trainieren und die gesamte Vielfalt der NLU-Phänomene abzudecken. Diese Herausforderungen unterstreichen den ständigen Bedarf an neuen Ressourcen und Datensätzen im Bereich des semantischen Parsings. Zudem sind neue Lern- und Evaluierungsverfahren gefragt, um Modelle robuster und anpassungsfähiger zu machen. Davon können wiederum Anwendungen profitieren, in denen verlässliche semantische Parser notwendig sind.

Das Problem der Datenknappheit bildet die Motivation für eine Verwendung von Programmcode-Bibliotheken als Parallelkorpora – also Sammlungen von Quellcode in Programmiersprachen mit der darin enthaltenen natürlichsprachlichen Dokumentation. Wir präsentieren 45 neue Datensätze in dieser Domäne- als Grundlage für die neuartige, anspruchsvolle Aufgabenstellung der Text-in-Code-Übersetzung (text-to-code). Zusammem mit weiteren Standarddatensätzen verwenden wir dieses Korpus dazu, semantische Parser für verschiedene Domänen

und Sprachen zu trainieren. Wir lösen dadurch neu entstandene Aufgaben wie die sprachübergreifende Dekodierung (polyglot decoding) zwischen verschiedenen natürlichen und formalen Sprachen sowie die Zero-Shot-Übersetzung (zero-shot translation) zwischen ungesehenen Sprachpaaren. Speziell für den Umgang mit unzureichender Datenmengen setzen wir ein neues Lernverfahren, welches Entailment-Information als ein schwaches Signal für das Training eines semantischen Parsers ausnutzt: Die Information darüber, ob die Bedeutung eines natürlichsprachlichen Satzes S aus der eines anderen Satzes folgt, wird verwendet, um über Lücken in der bisherigen Analyse des Parsers zu schlussfolgern und auf dieser Basis bessere semantische Repräsentationen zu lernen.

Der Gesamtbeitrag dieser Dissertation umfasst eine Vielzahl an neuen Methoden und technischen Lösungen für das Training von semantischen Parsern mit einen Minimum an Überwachung und manueller Anpassung. Dieser Ansatz hilft, die zu Beginn beschriebene Ressourcenknappheit zu umgehen. Des Weiteren stellen wir eine Reihe von neuen NLU-Problemstellungen zur Evaluation semantischer Parser vor. Wir glauben, dass dadurch die Mächtigkeit und praktische Verwendbarkeit von semantischen Parsern und NLU im Allgemeinen maßgeblich verbessert werden kann.

Acknowledgements

First and foremost, thanks to my *Doktorvater* Jonas Kuhn for all of his support and guidance through this whole thesis process. Without his patience, excitement and encouragement to explore new research ideas, this thesis certainly wouldn't exist. I am also indebted to my committee members: Dr. Jonathan Berant and Prof. Dr. Ulrich Hertrampf; thank you for your participation at the end of the thesis, and (to Jonathan) for your general feedback, encouragement and support at different critical stages of the thesis!

Thanks also to my colleagues and friends/family around the world for providing a wide range of support over the years: Anders Björkelund, Andre Blessing, the late Danny Bobrow (one of my earliest mentors in computer science), Cleo Condoravdi (my first boss in NLP), Özlem Çetinoğlu, Kerstin Eckart, Markus Gärtner, Abhijeet Gupta, Agnieszka Faleńska, Alex Fraser, Diego Frassinelli, Lauri Karttunen (another early mentor from my time at PARC), Maryna Kavalenka, Wiltrud Kessler, Roman Klinger, Maximilian Köper, Mirella Lapata (thanks for supporting my stay in Edinburgh at the end of the thesis, and for your general support and enthusiasm), Gabriella Lapesa, Florian Laws, Andreas Maletti, Lukas Michelbacher, Sabine Mohr, Sebastian Padó, Daniel Quernheim, Arndt Riester, Nils Reiter, Donald, Deborah and Benjamin Richardson, Michael Roth, Ina Rösiger, Wolgang Seeker, Christian Scheible, Lenhart Schubert (the person responsible for my initial interest in NLP), Asuman Sünbül, Jason Utt, Richard Waldinger, Xiang Yu, Annie Zaenen (another early mentor from PARC), Alessandra Zarcone and Sina Zarrieß. To my IMS colleagues in particular, thanks for creating such a pleasant and lively work atmosphere. Special thanks to Markus Gärtner, Joel Weller, Maryna Kavalenka, Sean Papay, Christian Scheible and Özlem Çetinoğlu for helping to proof various portions of the thesis.

This thesis is dedicated to Maryna and Leon.

Support: The work presented in this thesis was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich 732 "Incremental Specification in Context", project D2.

1 Introduction to Natural Language Understanding

1.1 The Problem of Natural Language Understanding

In this thesis, we investigate the general topic of *Computational Natural Language Understanding*, which has as its goal the development of algorithms and other computational methods that facilitate reasoning about natural language by the computer. The extent to which the computer reasons or thinks about language is determined by its ability to solve various natural language tasks, a small subset of which we examine in this thesis and describe in this introduction. Under such an approach, we subject the computer to various linguistic tests that probe the computer's knowledge of language and the world, knowledge that we assume is a prerequisite for solving each problem.

One such test that has intimate connections to the methods investigated in this thesis is automated question-answering, where the goal is to create programs that can retrieve answers to questions posed in ordinary natural language. For example, imagine that we have a background database such as the one in Figure 1.1 about flight schedules and we encounter the following question (Allen, 1987):

Which flights to Chicago leave at 4PM?

The task for our question-answerer is to return all flights that satisfy the constraints expressed in the query. As we discuss in this chapter, this problem can be broken down into several smaller sub-problems, and requires several levels of knowledge, some that go beyond the linguistic level. Chief among them is knowledge about how to translate this input query to an unambiguous formal representation with which the computer can reason exactly, in this case with the help of other downstream components such as a database management system or inference engine. Such a formal representation might take the following form:

flight
f105677
f105678
f105691
...

arrive
(f105677, "chicago", 18:30)
(f105678, "boston", 19:30)
(f105691, "london", 10:30)
...

depart
(f105677, "miami", 16:00)
(f105678, "moscow", 10:30)
(f105691, "berlin", 11:30)
...

Figure 1.1: An example airline database for querying.

```
(print-all ?f (and (flight ?f)
                   (arrive ?f 'chicago' ?t)
                   (depart ?f ?s 16:00)))
```

where each symbolic expression `(symbol` $\texttt{arg}_1$`,...)` maps either to some table in the example database or a procedure in a database program or logical inference engine. The final computation for finding the correct answer then requires substituting each variable $?v$ with values in the database that satisfy the constraints of the formal query and returning the value of the variable $?f$. In this case, the system should minimally return the flight identified as `f105677`.

As this example shows, by doing such a translation we are eliminating the ambiguity of ordinary language and making it possible to interface natural language with more complex systems and modalities. Thus, the ability to robustly translate natural languages to formal languages and representations in different domains, which we henceforth refer to as *Semantic Parsing* (SP), is of central interest to natural language understanding (NLU). This thesis focuses on the underlying algorithmic properties underpinning SP, and asks a general empirical question related to the practical learnability of NLU models of this type:

- **Learnability and NLU**: To what extent can we get the computer to learn natural language understanding models and semantic parsers from examples and minimal amounts of training supervision?

With this question, several new questions arise, such as the question of the types of *examples* we expect the computer to learn from, what *minimal* means in this case, and how we expect to find or collect such examples. The goal for this chapter is to explore these questions in more detail. To start the discussion, we further

motivate the *symbolic NLU* approach introduced above and give a brief overview of its origins in linguistics and artificial intelligence research. While reviewing this background, we identify more concretely the questions about learnability that are of interest to our work and introduce the following two learning settings: 1) learning from denotation and entailment and 2) learning from logical form.

1.1.1 Natural Language Understanding à la Montague

> *"There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematically precise theory."*
>
> – Richard Montague (1970) *Universal Grammar*

The formal study of natural language understanding in linguistics has its origins in the work of the mathematician Richard Montague, who advocated using the tools of mathematical logic to study natural language syntax and semantics. As the quote above indicates, he argued for studying natural languages with the same rigor as one studies the formal languages of mathematics and logic, and rejected any principled distinction between both types of languages.

While a full review of Montague semantics is beyond the scope of this chapter (for a computational overview, see Hobbs and Rosenschein (1977)), we consider the processing pipeline involved in his model of NLU. As shown in Figure 1.2a, for a given textual input, one starts by performing a syntactic analysis, which is then translated to a logical representation, specifically, a higher-order intensional modal logic, and interpreted semantically using semantic models. While his work predates much of the modern work in programming language theory, such a division of labor bears some resemblance to how modern computer compilers are designed and implemented (see Figure 1.2b), or computer programs that translate high-level programming languages to lower-level languages.

Pushing the analogy with programming languages further, we can think of a NLU system in the Montague tradition as a kind of compiler that translates natural language input (i.e., a high-level language) to lower-level programs that can be used for exact reasoning or execution. The connection between NLU and compiler design

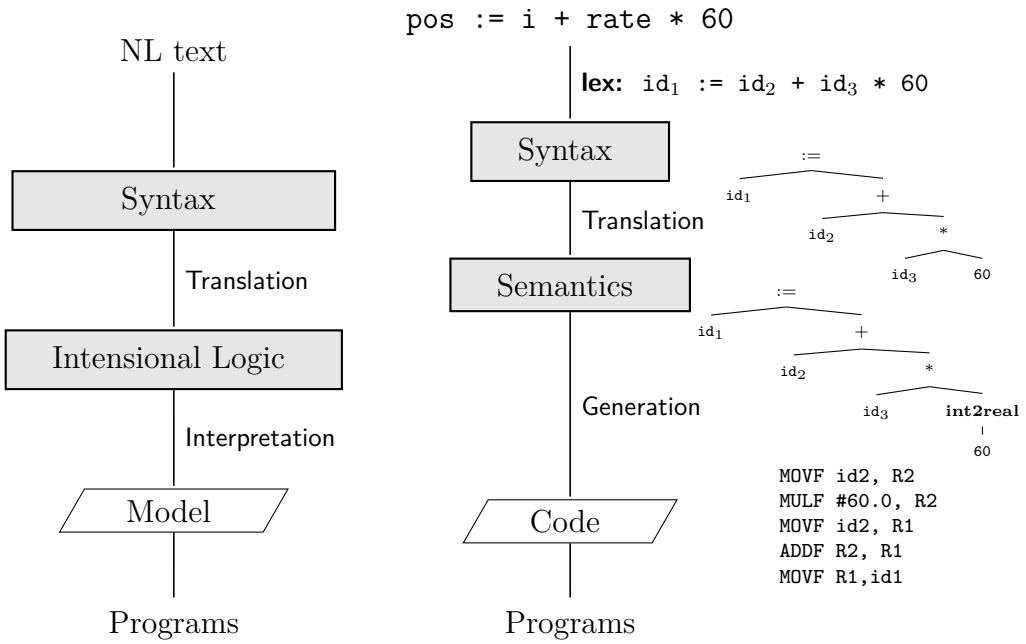


Figure 1.2: The Montague model of semantics (**a**) from Halvorsen (1986) and a standard compiler processing pipeline in (**b**) from Aho et al. (1986).

is not only conceptual, since many of the underlying methods, from the level of syntactic analysis to translation and semantics, rely on the same mathematics and set of algorithms. Formally, the goal is to model some *transduction* between a high-level input language $\mathcal{L}_{in}$ and output language $\mathcal{L}_{out}$, which we can define as some subset of $\mathcal{L}_{in} \times \mathcal{L}_{out}$ (Harrison, 1978). The way Montague defined such a mapping to logical languages, for example, works similar to how syntax-directed translators are defined in compilers (e.g., see Knuth (1968); Aho and Ullman (1969)), and his use of model theory and the lambda calculus to define a denotational semantics for a fragment of English mirrors how one ordinarily defines a denotational semantics for programming languages (Allison, 1986).

Beyond the technical contributions made by Montague, perhaps his most lasting contribution is methodological in nature, and involves his use of truth-conditions and entailment judgements as the *primary data* for semantics research. In Mon-

tague (1970), he describes the basic aim of semantics as the ability 'to characterize the notion of a true sentence (under a given interpretation) and of entailment.' In practice, when developing theories of semantics, semanticists use these judgements to check that their resulting semantic models account for all known facts about truth and entailment, and modify their assumptions and theories when certain facts are not captured. Such a focus on truth-conditions and entailment relations greatly expanded the *adequacy criteria* for semantics theories and the breadth of research (Partee, 2005), in a way similar to Chomsky (1965)'s use of judgements about grammaticality in syntax research.

It is instructive to consider how this approach to semantics works in practice. As an illustration, consider the following linguistic example from Donald Davidson's work on event semantics (Davidson, 2001):

Jones buttered the toast slowly ... in the bathroom with a knife... (1.1)

which he uses to ask the question: how should the logical form for these types of action sentences look like? The specific difficulty is how to represent the adverbial modifiers *slowly, in the bathroom,...* while accounting for the following entailments (marked using the $\models$ symbol) 1.3-1.8 from 1.2:

Jones buttered the toast slowly in the bathroom with a knife (1.2)

(1.2) $\models$ Jones buttered the toast slowly in the bathroom (1.3)

(1.2) $\models$ Jones buttered the toast slowly (1.4)

(1.2) $\models$ Jones buttered the toast (1.5)

(1.2) $\models$ Jones did something that was in the bathroom (1.6)

(1.2) $\models$ The thing that Jones was involved with was slow (1.7)

(1.2) $\models$ The thing that Jones was involved with involved a knife (1.8)

On the basis of this evidence (i.e., the entailments), it is insufficient to represent each adverbial modifier as a slot in the main predicate `butter` as shown in 1.9. This approach also has the downside of requiring a large number of predicates with a variable number of argument slots. The general observation is that one can drop

```
(define butter-denot '(("jones" "toast" "e1")
                       ("john" "cracker" "e2")))
;; the denotation of butter, or ⟦butter⟧
(define slow-denot '("e1")) ;; ⟦slowly⟧

(define butter (lambda (s)
                 (lambda (o)
                   (lambda (e)
                     (contained-in (list s o e)
                                   butter-denot)))))
;; => λs.λo.λe. butter(s,o,e), in curried form

(define slowly (lambda (e) (contained-in e slow-denot)))
;; => λe.slowly(e)

(((butter "jones") "toast") "e1")
;; => λo.λe.butter("jones",o,e) => λe.butter("jones","toast",e)... => True
(slowly "e2") ;; => False
```

Figure 1.3: An example functional interpreter for the Davidson fragment implemented in the Lisp programming language.

the modifiers freely in this case without affecting the entailment relations, which therefore requires representing each modifier separately from the main predicate. For this reason, Donaldson argues for the existence of events and event entities, e, and offers the logical form in 1.10:

$$\texttt{butter(jones,toast,slowly,in\_bathroom,...)} \tag{1.9}$$

$$\exists \texttt{e. butter(jones,toast,e)} \wedge \texttt{slowly(e)} \wedge \texttt{with(e,knife)} \wedge \texttt{...} \tag{1.10}$$

where the event variable e captures the general eventuality being described, which is referred to in 1.7-1.8 as *the thing that John was involved with*, and the modifiers are given as separate predicates over the event. Without this new variable and extra level of abstraction, according to Davidson's argument, the correct entailments do

not hold, which therefore makes the alternative analysis in 1.9 untenable. Using the analogy with programming languages once more, we can think of these tests of entailment as analogous to unit testing in software development, or when one subjects one's program semantics or code to example input and output to check that the program behaves as expected. If such a test fails, then the programmer has made a mistake that needs to be fixed, which might similarly involve introducing more levels of abstraction into the associated program.

Given the formal connection between semantic theories in the Montague tradition and the theories of computation and programming discussed above, the practical implementation of such theories and fragments is well understood and details can be found in several textbooks on the subject that commonly use techniques from either logic programming (Blackburn and Bos, 2005) or functional programming (Gazdar and Mellish, 1989; Allen, 1987; Van Eijck and Unger, 2010). An extensional, Montague style implementation and functional interpreter for the Davidson fragment is shown in Figure 1.3 using the Lisp language first introduced in McCarthy (1960), which, like Montague's theory, is based on the lambda calculus of Church (1932).

In all of the work cited above, computational semantics (i.e., the study of the computational implementation of semantic theories) works top-down from linguistic theories about representations, such as the analysis of events already introduced, to explicit implementations or coding of these theories, such as the implementation in Figure 1.3. Van Eijck and Unger (2010) specifically describe two main uses for machines in computational semantics, first to automate the construction of meaning representations (largely with the help of linguist programmers), and second to perform deductive inference on these generated representations. In this thesis, we consider a third use of machines in this endeavor, which centers around the inductive learning of semantic theories using machine learning.

One of the learning settings that we investigate in this thesis (Chapter 5) is based on the following question:

1. **Learning from Denotation and Entailment**: Rather than starting with an explicit implementation for a natural language fragment or set of data, can we teach the computer to learn backwards from denotations and entailments

to the correct representations?

For example, rather than translating our complete theory of events to code, we might start by giving the computer the data in 1.1-1.8 and the deficient semantic analysis in 1.9. By providing the computer with some additional facts about logic and the target domain, we then can train the computer to find the correct representations, within some constrained space of possible representations, that leads to the correct entailments through some form of trial and error. In other words, we want to train the computer to behave like a semanticist, and to ground decisions about representations by working backwards from the target data. In this case, the aim is to teach the computer to break the representation in 1.9 into more local modifier predications. To test if the goal has been accomplished, we can subject the computer to new examples to see if it makes the correct decisions.

The general motivation for this type of experimentation, which has gained some traction in modern natural language processing (NLP) (Clarke et al. (2010); Artzi and Zettlemoyer (2011); Liang et al. (2013); Berant et al. (2013)), is to see how much the computer can learn about language and the world from minimal amounts of explicit instructions, a motivation that is of more general scientific interest, especially to the field of machine learning. A more practical motivation for this type of automatic theory construction is to make it easier to design and implement semantic theories by bootstrapping from data. As we discuss in the next subsection, the engineering of such semantic models and knowledge resources has proven to be a major bottleneck in AI research on NLU, which makes new methods that replace the need for hand-engineering a valuable resource.

1.1.2 The View from AI and NLP

> *"I was searching for a method of semantic interpretation that would be independent of particular assumptions about data base structure... The method I developed was essentially an interpretation of Carnap's notion of truth conditions (Carnap, 1964a). I chose to represent those truth conditions by formal procedures that could be executed by a machine.... This notion, which I referred to as 'procedural semantics,' picks up the chain of semantic specification from the philosophers at the level of abstract*

> *truth conditions, and carries it to a formal specification of those truth conditions as procedures in a computer language"*
>
> – William Woods (1978)

In classical AI and NLP research, one can find strong similarities between the standard approach to NLU and the Montague program, even in work that predates Montague's seminal work (e.g., Bobrow (1964)). For most of AI's relatively short history, there has also been a strong focus on logic and reasoning, or symbolic AI. This largely stems from early results on the limitations of rival connectionist (i.e., non-symbolic) models, most notably the work by Minsky and Papert (1969) on the limitations of the single-layer perceptron model of Rosenblatt (1958). In contrast to work in linguistics, however, much of this early research centers on concrete applications such as automated question-answering and robot planning, among others. The applied character of this early work is captured in the quote above by William Woods when he talks about reinterpreting the philosopher's or logician's abstract notion of truth in terms of formal procedures that can be executed by a machine. This is a framework for semantics that he and others from that time refer to as *procedural semantics* (Johnson-Laird, 1977; Woods, 1968).

An illustration of the LUNAR question-answering system of Woods (1973), which is an example of this procedural approach, is shown in Figure 1.4. While his system was not the earliest attempt at automated question-answering (for a review of earlier attempts, see Simmons (1965) and Bobrow (1964)), it is of some historical interest since, as Woods mentions above, his approach was perhaps the first to be based on a more general theory; one in which the model of syntax and semantics was not directly tied to the structure of the target data and domain. Similar to the processing pipeline for Montague semantics introduced previously, the system starts by performing a translation to a logical form, where predicates are defined as executable Lisp procedures, and answers are retrieved by executing these formal representations against a database, which plays the role of a *world* or *universe of discourse* in more abstract theories of semantics.

At a lower level, Woods describes the pipeline model shown in Figure 1.5, where he additionally assumes a syntactic component, which at the time was based on his well-known work on context-free parsing using augmented transition network

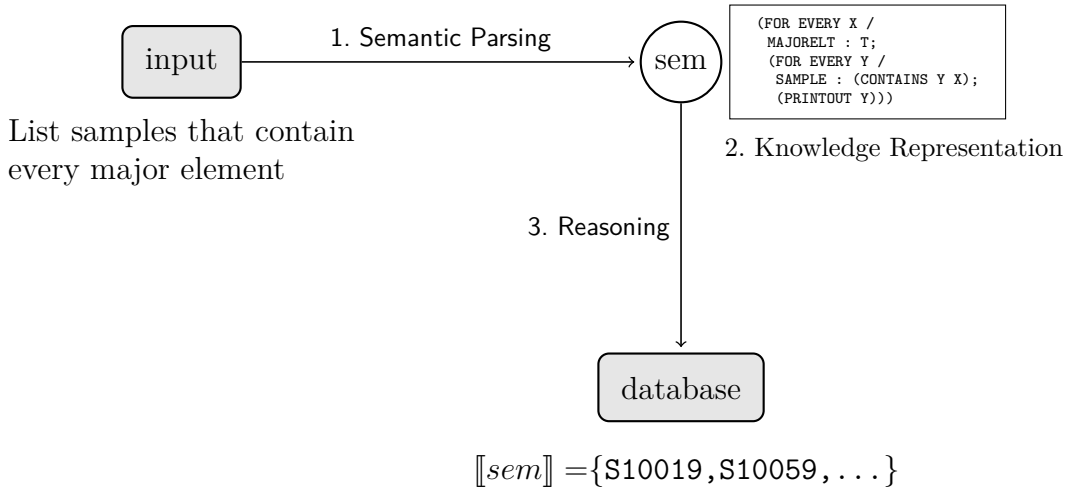


Figure 1.4: An illustration of the LUNAR question-answering system and the different tasks involved in NLU.

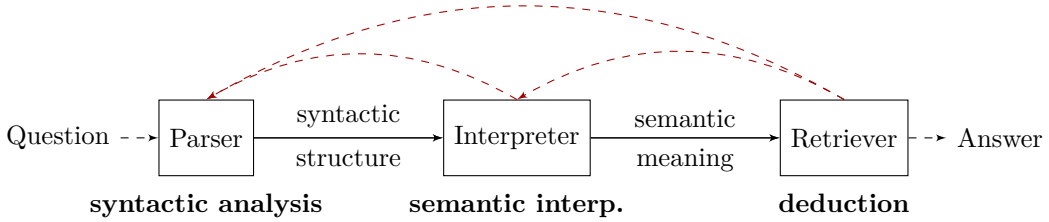


Figure 1.5: A more detailed processing pipeline in the Woods model.

grammars (Woods, 1970). While this approach appears to closely resemble the compiler model already introduced, one additional detail is the interaction between the different levels of processing, as shown using the red dashed lines. The idea is that the semantic or retrieval component might provide feedback to the syntactic component when certain structural or semantic (or maybe even pragmatic) ambiguities emerge, which is an idea that we return to in the next section.

Woods (1968, 1978) provides the technical details for the interpretation component in his models, which maps syntactic structure to semantic representations using a set of translation rules defined on top of these syntactic patterns. So far, our discussion of such components has been high-level, so we consider one particu-

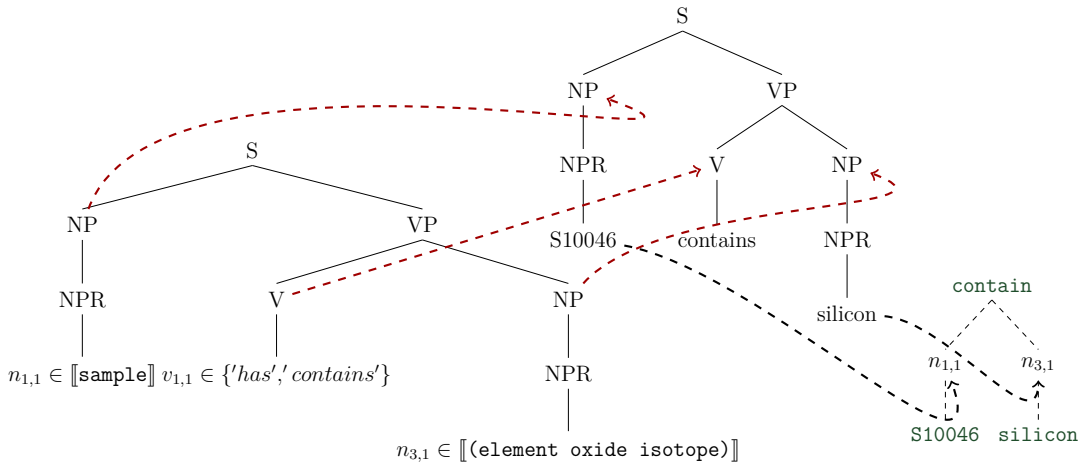


Figure 1.6: An illustration of the semantic interpretation mechanism in the Woods model for the input *S10046 contains silicon*.

lar example below, which Woods calls a *pattern* → *action* rule. Using a Lisp style notation, the left hand side of the rule is a syntactic pattern to be matched, and the right hand side shows the resulting semantic rule:

```
[S:CONTAIN
        (S.NP (MEM 1 SAMPLE))
        (S.V (OR (EQU 1 HAVE)
                 (EQU 1 CONTAIN)))
        (S.OBJ (MEM 1 (ELEMENT OXIDE ISOTOPE)))
        -> (QUOTE (CONTAIN (# 1 1) (# 3 1)))]
```

Using a more modern presentation of this idea, we can describe this rule using tree transformations as shown in Figure 1.6. Here, the first tree describes the abstract rule template, where the terminating nodes in the tree specify general rules about the semantic type of the target terminating nodes in an input tree. For example, the first subject `NP` should contain a word that is a type of `sample`, and the main verb in the `VP` should be either the word *has* or *contains*. The second tree shows a parse tree that matches these set of rules, and the third tree is the resulting

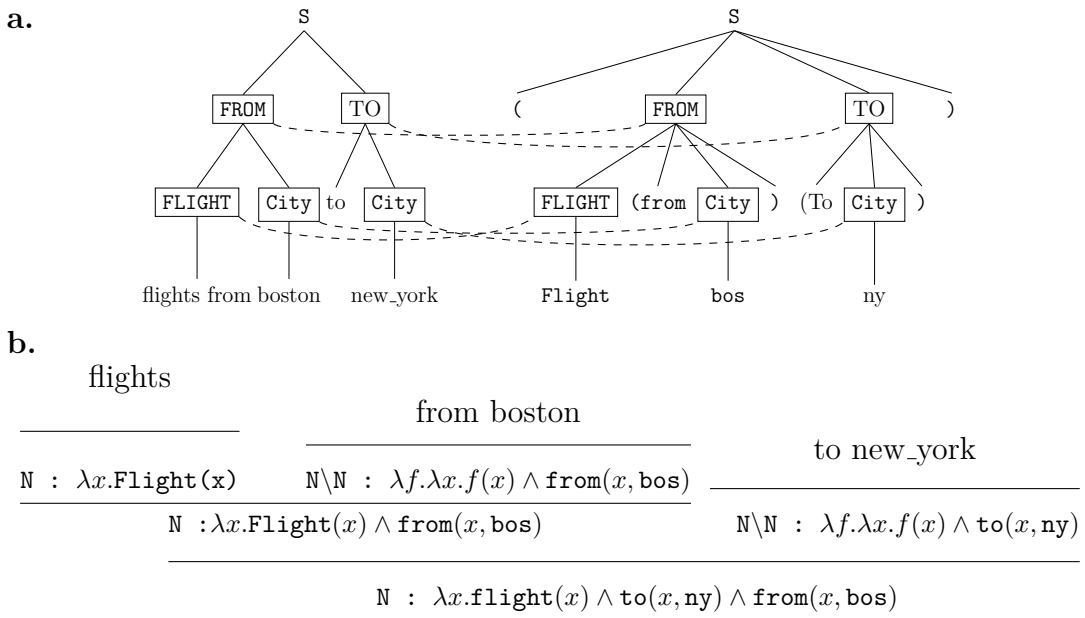


Figure 1.7: An illustration of two approaches to syntax-directed translation based on synchronous grammar (a) and categorial grammar (b) in the ATIS domain (Dahl et al., 1994).

semantic rule, which transforms the matching tree into the logical representation `(contains S10046 silicon)`.

When viewed in this way, his semantic interpreter has the appearance of a syntax-based machine translation model (Williams et al., 2016). Under a translation approach, we can regard the output semantic representation as itself being a language, an idea that we pursue throughout this thesis. Modern incarnations of these models rely on formal methods that were not widely studied in NLP during the time of of Woods' work, such as tree automata and grammars, first looked at by Rounds (1970) for developing formal models of transformational grammars and later by researchers in theoretical computer science (Comon et al., 2007), as well as related methods such as synchronous grammars (Shieber and Schabes, 1990), term rewriting (Baader and Nipkow, 1999), categorial grammars (Steedman, 1996), among many others (see Figure 1.7 for an illustration).

In terms of the practical implementation of these AI models, much subsequent

work has focused on explicit hard-coding of the type of semantic interpretation or *transfer* rules (Dorna and Emele, 1996) considered above. This includes work in the context of large scale grammar development efforts, such as the LFG-based ParGram project (Butt et al., 2002) and work on the LFG semantic transfer system (Crouch, 2005; Crouch and King, 2006), and similar efforts in the HPSG community (Copestake and Flickinger, 2000) on implementing minimal-recursion semantics (Copestake et al., 2005). Another large-scale attempt at domain-general NLU is work on the Boxer system (Bos, 2008), which implements a large fragment of Discourse Representation Theory (Kamp and Reyle, 2013). Outside of these large-scale efforts, the hand-engineering of semantic translation rules is still commonly pursued for small-scale application building (Popescu et al., 2003; Unger et al., 2012; Waldinger et al., 2011; Condoravdi et al., 2015).

Recalling again our focus on learnability, we ask the following question related more to the problem of learning semantic representations:

2. **Learning from Logical Form**: Rather than hand-coding semantic translation rules, can we learn semantic translations rules from parallel data consisting of unaligned input text and the semantic representations?

For example, rather than hand-coding the rules described in Figure 1.6, we might provide the computer with the input text *S10046 contains silicon*, with or without its associated syntax tree, and the target semantic representation `(contains S10046 silicon)`. The computer is then expected to learn a hidden, or *latent*, mapping between the two representations, which may or may not resemble the rules considered previously. The ultimate test is then to measure the accuracy with which the computer can generate correct representations for unseen examples, where the correctness of a representation can be evaluated in one of two ways:

- **Intrinsic Evaluation:** Does this representation match a gold example, one that is generated by a human or appears correct to a human?
- **Extrinsic Evaluation:** Can the representation be used to solve a downstream task: one in which the expected outcome is achieved?

It is important to note that the evaluation methods listed above require different amounts of human effort, which is an issue that motivates some of the decisions made in our work. Doing an intrinsic evaluation on a question-answering system such as Woods' requires annotators to create *gold* annotations or representations. This in turn requires finding annotators with considerable knowledge of logic or database theory, not to mention expertise in the target domain. Another downside is that errors in the human annotation might in the end unfairly punish the machine's performance on the evaluation. In the extrinsic case, experts need only provide the answers to the target questions, which removes the need for difficult annotation, and might better reflect the machine's performance since the machine is free to find an alternative translation to the annotator's.

Work on learning from logical forms, within the subfield known as *data-driven semantic parsing*, is fairly new and was largely motivated by the difficulties of hand-engineering the types of semantic translation models cited above. As discussed in Mooney (2007b), these approaches are explicitly in contrast to work on learning intermediate or shallow semantic representations, such as semantic role-labeling (Gildea and Jurafsky, 2002) or geometric/distributional approaches to semantics (Widdows, 2004; Turney and Pantel, 2010). The ultimate goal of semantic parsing is to learn *complete*, formal representations that resemble the classical logical semantic representations discussed above. Early attempts looked at learning transformation rules and learning from logical forms using somewhat ad hoc formal methods (Kate et al., 2005), and later tools primarily from statistical machine translation (SMT) and statistical parsing. We provide a brief summary of the different approaches in Table 1.1, with self citations shown in red. At the time of writing, there has been increased interest in approaches based on neural sequence to sequence modeling (Sutskever et al., 2014) (see last row of Table 1.1), following a more general embrace of deep learning techniques in NLP.

One important methodological assumption inherent in work on data-driven semantic parsing is that the underlying methods being investigated and developed should be generalizable, an idea that we can describe in the following way:

- **Domain agnostic constraint:** The underlying semantic parsing and translation methods should not be tied to any one target domain; they should be

Approach/Framework	References
Combinatory Categorial Grammar	Zettlemoyer and Collins (2009, 2012); Kwiatkowski et al. (2010); Krishnamurthy and Mitchell (2012); Artzi and Zettlemoyer (2013); Artzi et al. (2015); Kushman and Barzilay (2013); Reddy et al. (2014)
Context-Free Grammar (CFG)	Angeli et al. (2012); Börschinger et al. (2011); Liang et al. (2013); Berant et al. (2013); Berant and Liang (2015); Kim and Mooney (2012); Zhang et al. (2017); Richardson and Kuhn (2012, 2016)
Synchronous CFG/SMT	Wong and Mooney (2006, 2007); Li et al. (2015, 2013); Andreas et al. (2013); Arthur et al. (2015); Haas and Riezler (2016); Richardson and Kuhn (2017b,a); Zarrieß and Richardson (2013)
Tree/Graph Grammars and Related	Jones et al. (2012b,a); Quernheim and Knight (2012); Koller (2015); Peng et al. (2015); Chiang et al. (2018); Groschwitz et al. (2015)
Neural/RNN Approaches	Dong and Lapata (2016); Jia and Liang (2016); Kočiský et al. (2016); Herzig and Berant (2017); Cheng et al. (2017); Krishnamurthy et al. (2017); Duong et al. (2017); Richardson et al. (2018)

Table 1.1: A brief survey of the various semantic parsing approaches from the literature (with self citations in red).

applicable to new domains and semantic language types.

For example, when investigating semantic parsing in the domain of Lunar geology, which is the domain investigated by Woods, the underlying methods that generate semantic representations from text should work equally well when applied to other domains, such as mapping to representations in the airline planning domain first introduced or for non-procedural knowledge representation types. This constraint is clearly violated in the case of Woods' approach to semantic parsing, since the interpretation rules that he devises, such as the one in Figure 1.6, are narrowly tied to the domain of Lunary geology and cannot be ported to other domains. As we discuss in the next section, the inability of methods in NLU to generalize has perhaps been a main reason for why the classical methods have had limited success in mainstream and commercial NLP.

Sub-problem	Problem Description
1. Semantic Parsing (SP)	*Translating* `input` *to* `sem`, `input` $\rightarrow$ `sem`
2. Knowledge Representation	*Defining a sufficiently expressive* `sem` *language.*
3. Reasoning and Execution	Going from `sem` to denotations and data

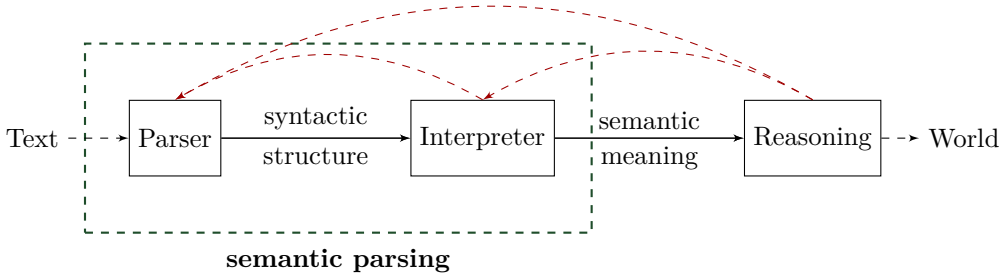


Figure 1.8: A definition of the different sub-problems involved in NLU and the relation to the Woods model below.

1.2 Towards a Modular Data-driven Approach

In this section, we describe our approach to NLU based on learnability, starting with an examination of the different sub-tasks involved in our model and a discussion of data resources for semantic parsing.

1.2.1 Defining the Sub-Tasks

1. Semantic Parsing Bringing together the ideas discussed in the previous section, we break down the different tasks in NLU to those shown in Figure 1.8, which is a somewhat simplified version of Woods' description in Figure 1.5. The first task, of primary interest to this thesis, is semantic parsing, which concerns the task of translating input text to semantic representations. More specifically, we are interested in learning these translations from data, a problem that we can describe more concretely as follows:

- **Data-driven Semantic Parser Induction**: Given a parallel training dataset D consisting of text-meaning pairs, learn a function that maps any given input text $\mathbf{x}$ to its correct semantic representation $\mathbf{z}$, or $\mathtt{sp} : \mathbf{x} \rightarrow \mathbf{z}$

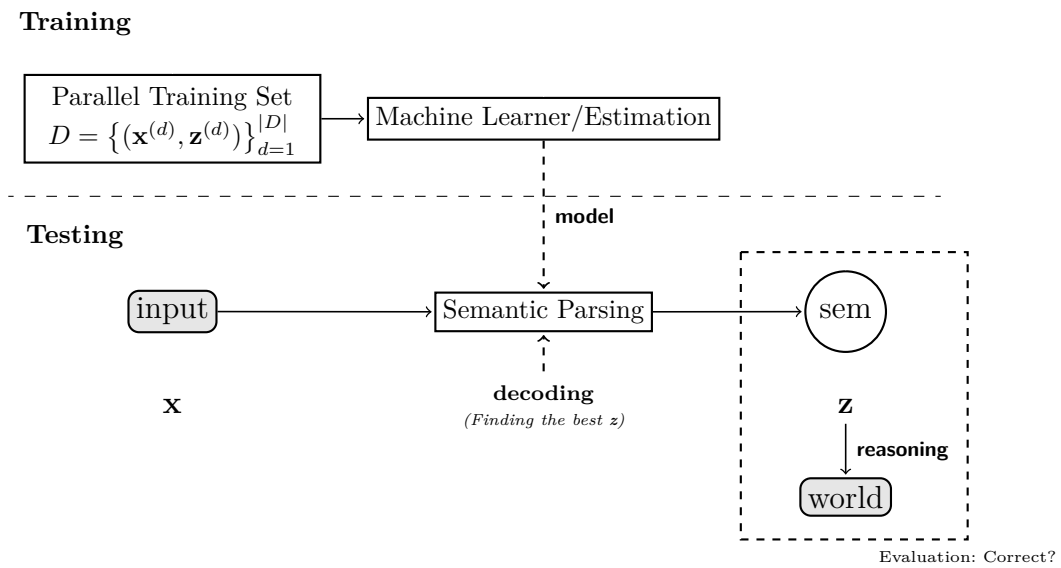


Figure 1.9: An illustration of the machine learning setup for data-driven semantic parser induction.

When doing data-driven semantic parser induction, we will assume a rather conventional machine learning setup as illustrated in Figure 1.9. During a *training* phase, the goal is to train a machine learning model using parallel data consisting of example input and output pairs. Such a model is then used at *testing* time in combination with a decoding or search algorithm to generate semantic output from input. In an experimental setting, we can then evaluate the resulting output using either an intrinsic or extrinsic evaluation as discussed in the previous section.

Extending our notion of a semantic parser as formally modeling some transduction, in the data-driven case, the goal is to model some weighted transduction or weighted relation:

$$\texttt{sp} : \Big(\mathcal{L}_{\texttt{in}} \times \mathcal{L}_{\texttt{out}}\Big) \to \mathbb{R}. \tag{1.11}$$

Throughout this thesis, we will be interested in types of *conditional probabilistic relations* that we can estimate using a statistical model that will often take the form

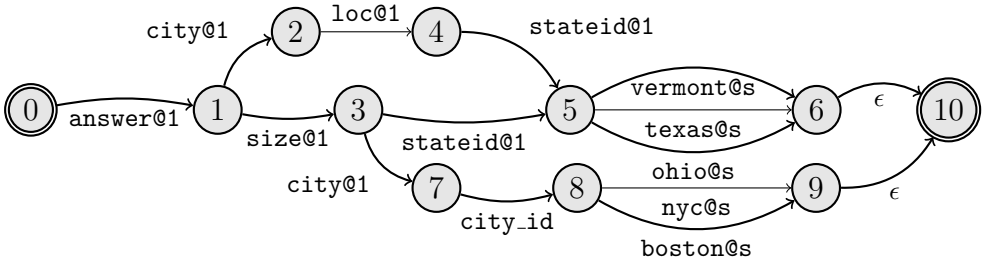


Figure 1.10: A DAG representation of the search space for a fragment of the Geoquery language from Andreas et al. (2013).

of a conditional probability distribution: $p(\mathbf{z} \mid \mathbf{x})$. While we investigate several such models throughout the thesis, all involve solving the following general problems (as shown in Figure 1.9):

- **Estimation**: How do we find the optimal parameters of a given model using example *parallel data* (i.e., the example inputs and outputs encountered during the training phase)?
- **Decoding**: Given a model and an input $\mathbf{x}$, how do we generate the best output $\mathbf{z}^*$ (or k-best outputs) within $\mathcal{L}_{out}$? This will often involve solving:

$$\mathbf{z}^* = \arg\max_{\mathbf{z} \in \mathcal{L}_{out}} \Big\{ p(\mathbf{z} \mid \mathbf{x}) \Big\}.$$

Given that semantic parsing involves translating to formal languages, most models, despite their differences, make the assumption that the output languages being learned and generated are highly structured. One theme throughout this thesis is using static graph representations, such as the one shown in Figure 1.10, to represent the search space of semantic parsers (where each path in the graph is associated with a possible translation). Under such a graph approach, estimation can be described as finding a model that optimally associates correct paths (i.e., translations) with training inputs $\mathbf{x}$. Decoding is then the problem of finding the optimal path (within a large space of possible paths and translations) given an estimated model and particular input $\mathbf{x}$.

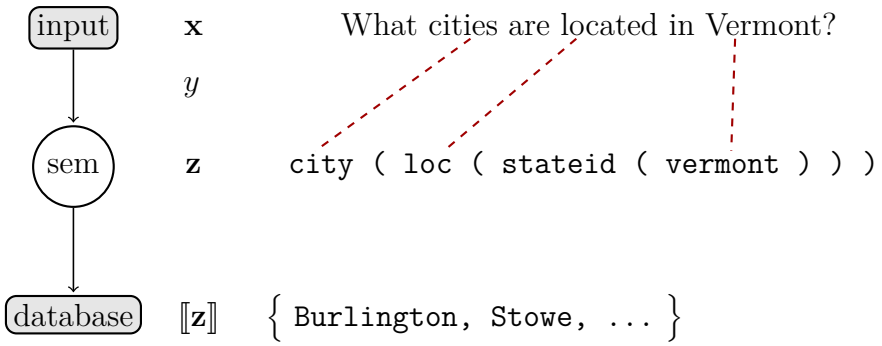


Figure 1.11: Example semantic parsing data from the Geoquery domain.

Paradigm and Supervision	**Dataset $D =$**	**Learning Goal**
Learning from logical form	$\{(input^{(d)}, LF^{(d)})\}_{d=1}^{\|D\|}$	$input \xrightarrow{Trans.} LF$
Learning from denotation	$\{(input^{(d)}, [\![LF^{(d)}]\!])\}_{d=1}^{\|D\|}$	$input \xrightarrow{LF+Trans.} [\![LF]\!]$

Figure 1.12: The different learning settings for semantic parser induction

To illustrate this, Figure 1.11 shows an example from one benchmark semantic parsing dataset called the Geoquery (Zelle and Mooney, 1996), which consists of pairs of English questions about American geography and formal Prolog database queries. The goal is to learn a function that can translate examples such as $\mathbf{x}$ to the representations in $\mathbf{z}$ (in this case, $\mathbf{x}$ to `city(loc(...( vermont )))` which is a single path in Figure 1.10). It is important to note that in the definition above, we only specified that the dataset D must be *parallel*, but we did not specify the form that *meaning* takes in such a parallel dataset. This is intentional, since different learning settings make different assumptions about the type of *supervision*, or evidence provided to the machine learner, that is needed to learn semantic parsers.

Using this new definition, we summarize in Figure 1.12 the different learning settings discussed in Sections 1.1.1-1.1.2. In the *learning from logical form* setting, we assume that D consists of example text input and fully annotated logical forms, such as the pair $(\mathbf{x}, \mathbf{z})$ from Figure 1.11. The learning problem in this setting then reduces to a translation problem, where the goal in each case is to learn a hidden translation y that correctly derives $\mathbf{z}$ from $\mathbf{x}$. In the *learning from denotation* or

Dataset	**Size** (# sentence pairs)	**Domain**
ATIS (Dahl et al., 1994)	5,410	Airline planning
GeoQuery (Zelle and Mooney, 1996)	880	Geography
Jobs (Tang and Mooney, 2000)	640	Job listings
Sportscaster (Chen and Mooney, 2008a)	1,872	Sports commentary
SAIL (Chen and Mooney, 2011)	3,233	Navigation instructions
Freebase917 (Cai and Yates, 2013)	917	open domain questions
WebQuestions (Berant et al., 2013)	5,810	open domain questions
WikiTableQA (Pasupat and Liang, 2015)	22,033	open domain questions

Table 1.2: Brief survey of benchmark semantic parsing datasets (red highlighting shows the datasets used in this thesis).

entailment setting, D consists of text and some representation of the semantic denotation of the text. In the Geoquery case, this would be the answer to the question, or { `Burlington, Stowe,..`} in Figure 1.11. Here, the learning task is considerably harder, since not only is the translation y a latent variable, but so is the correct logical form $\mathbf{z}$. While learning a translation is still involved, the larger learning problem takes the form of a program induction or program synthesis task (Liang et al., 2013).

Semantic Parsing Resources Similar to the discussion about evaluation in the last section, these different learning settings assume differing levels of human annotation effort. Learning from logical forms requires having a dataset of annotated logical forms, whereas learning from denotation requires having higher-level representations of meaning not necessarily tied to logic. Doing annotation for the latter type of learning is usually easier than the former, but at the cost of making the learning problem harder. Despite these differences, however, finding data of either type without involving considerable manual engineering effort is often one of the first bottlenecks encountered when building data-driven semantic parsers, a problem that we look closely at throughout the thesis and refer to as the *resource problem*:

- **The Resource Problem for Semantic Parsing**: How can we create parallel data for developing robust semantic parser methods with minimal engineering and annotation effort?

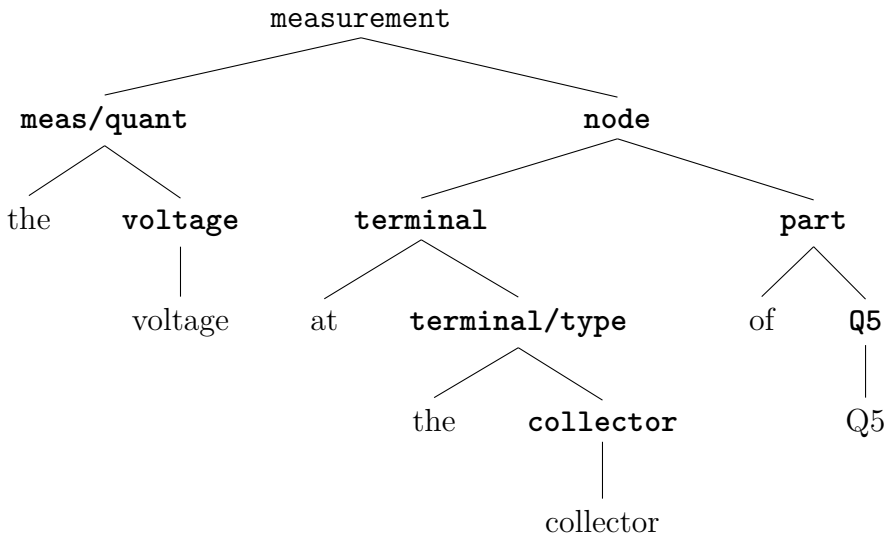


Figure 1.13: An example derivation in one of the earliest semantic parsers from Burton (1976) in the domain of electronics.

In the history of machine translation (MT) research, developments in statistical MT have been facilitated by the availability of naturally occurring parallel datasets largely coming from the domain of political proceedings in countries or regions where such proceedings are translated into multiple languages. Examples include the Canadian parliament proceedings (Gale and Church, 1993), and the European Parliament proceedings (Koehn, 2005). As Halevy et al. (2009) argue, the "biggest successes" in NLP have been in statistical speech recognition and statistical machine translation largely because "a large training set of input-output behavior that we seek to automate [in these tasks] is available to us in *in the wild*" (see discussion in Riezler (2014)). The essence of the resource problem for SP is that the main data of interest does not naturally occur in the wild. To date, most benchmark datasets tends to be limited in size and scope (see Table 1.2 for details).

Looking again at the sub-tasks listed in Figure 1.8, we point out that in our model, the semantic parsing task subsumes the task of syntactic analysis discussed previously. In other words, when translating to a logical form, a syntactic analysis of the source text is assumed to be part of the underlying translation process.

Studies differ in terms of how directly traditional syntax information is used; earlier studies assume access to a full syntactic parser (Ge and Mooney, 2005; Kate et al., 2005), whereas later studies use more domain-specific syntax representations based on *semantic grammars* (Wong and Mooney, 2006; Börschinger et al., 2011), or syntax models in which the non-terminal syntactic categories are replaced with semantic or conceptual categories (see Figure 1.13 for an example from Burton (1976)). Even later studies in the statistical machine translation tradition (Andreas et al., 2013; Jia and Liang, 2016; Dong and Lapata, 2016) treat both the input and output as linear sequences. At the time of writing, the latter approach has become dominant and is one of the primary approaches we pursue throughout this thesis.

Knowledge Representation (2) and Reasoning (3) Moving down the sub-problem pipeline, we have the problem of knowledge representation (problem number 2 in Figure 1.8), which concerns the study of the formal languages and representations we use to capture the target semantics. Given our *domain agnostic constraint*, this tends to receive less focus in the semantic parsing literature since the goal is to develop semantic parsing methods that can be applied to any representation type. Consequently, we often rely on the insights and best practices from linguistic semantics, which largely concerns itself with problems of knowledge representation and has traditionally relied on tools such a first-order logic. The same is true for the *reasoning* or *executation* problem (problem number 3 in Figure 1.8). When, say, building a natural language interface to a database, we assume the normal tools for querying the database once we have mapped to a formal representation. In some learning settings, however, understanding the interaction between the different sub-problems (shown in Figure 1.8 using the red lines) is important, a issue that we describe as follows:

- **Sub-task Interaction and Learning**: How do the different NLU sub-components interact under different learning settings?

When learning from logical forms, the interaction between the semantic parser and the decisions about knowledge representation are less connected, and both components can be studied in a modular fashion. However, when learning from denotation and other higher-level annotation types, the interaction between the

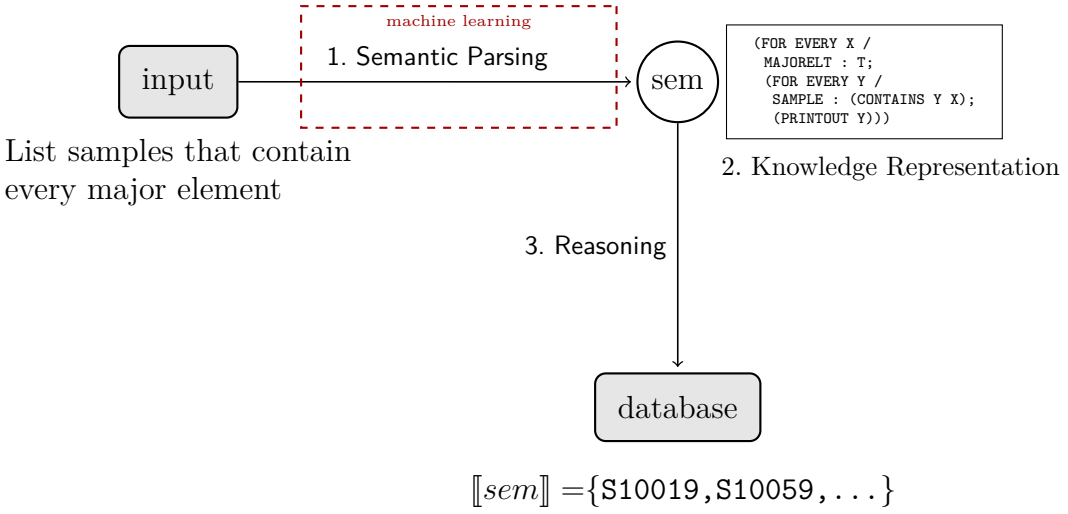


Figure 1.14: The basic model of NLU with machine learning added to the semantic parsing component (shown in red).

different levels is much more direct; since the target representation is latent in this case, more complex representations can seriously complicate the learning problem, a topic looked at in Liang et al. (2013).

1.2.2 Combining Statistical and Logical Semantics

> *"Machines and programs which attempt to answer English questions have existed for only about five years. But the desire to translate language statements into symbols which can be used in a calculus has existed as a long as formal logic. Attempts to build machines to test logical consistency date back to at least Roman Lull in the thirteenth century... Only in recent years have attempts been made to translate mechanically from English into logical formalisms"*
>
> – R.F. Simmons (1965) *Answering English Questions by Computer*

As described in the quote above (and throughout this chapter), semantic parsing is an old task that can be traced to the beginnings of AI research and logic. The dream of semantic parsing is to *mechanically* translate natural language text to

unambiguous formal languages with which one can reason exactly and interface with other modalities. Given that semantic parsing is largely placed within a symbolic approach to AI, the integration with a statistical approach has not always been obvious. Under the statistical approach, one works according to the following principle:

- **Statistical Semantics Hypothesis:** Statistical patterns of human word usage can be used to figure out what people mean. (Turney and Pantel, 2010)

In classical natural language understanding, it is not on the basis of statistical patterns of words alone that one arrives at an analysis of meaning, but rather through a translation from words to an *unobserved* formal language. Hence, it is the unobserved nature of this formal language that has created problems for integrating the two approaches.

The statistical approach to NLU and semantic parsing outlined in this chapter offers the following solution (as illustrated in Figure 1.14): by providing the computer with clues about these formal languages (either directly, by providing an explicit parallel corpus, or indirectly by providing information about denotations) we can then use statistical modeling to try to robustly translate between the two and in the end, capture the meaning in accordance with the classical approach to NLU. As described by Liang and Potts (2015), "the divide [between logical and statistical approaches] is rapidly eroding with the development of statistical [semantic parsing] models that learn *compositional semantic theories* from corpora and databases." Based on this approach, we can state the following hypothesis (as a version of the statistical semantics hypothesis above):

- **Statistical Semantic Parsing Hypothesis:** Statistical patterns of word usage paired with formal representations or other *grounded* formal systems can be used to figure out what people mean by learning the translation from text into these formal systems.

As already stated, one of the main bottlenecks in this approach is the *Resource Problem* outlined in the previous section (i.e., the problem of how we find these target formal representations and formal systems), which will figure rather centrally into the content of this thesis, and we integrate this problem into the list of sub-problems in Table 1.3.

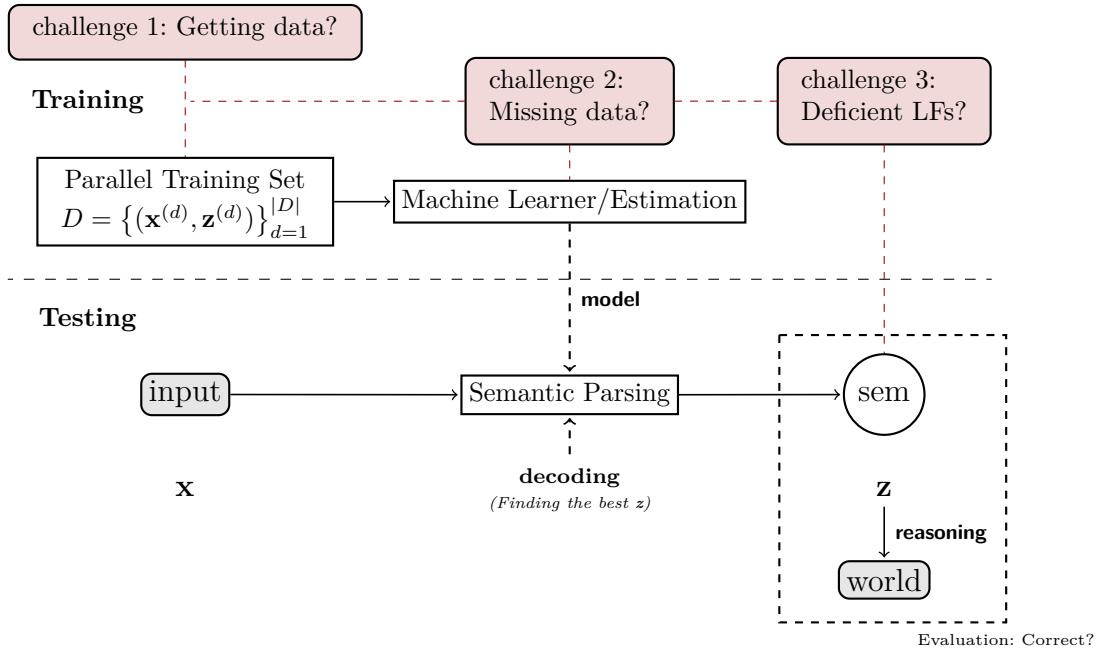


Figure 1.15: An illustration of the three resource issues addressed in this thesis.

1.3 Thematic Overview of Thesis and Contributions

In this thesis, we focus on three underlying challenges (all variants of the *resource issue* described previously) that one encounters when developing robust data-driven semantic parsers in the setting shown in Figure 1.9. These challenges are illustrated in Figure 1.15 and include: 1) the challenge of finding new parallel datasets given that semantic parsing data is traditionally expensive and time consuming to build and annotate; 2) the challenge of having missing data for a particular domain or having an insufficient amount of data for training robust semantic parsing models; and 3) the challenge of having parallel data in which the annotations are *deficient*, or fail to help us solve downstream NLU tasks.

As suggested by the title of this thesis, we focus on the following two overarching themes throughout the thesis:

- **New resources**: Finding new types of parallel datasets for benchmarking and developing new semantic parsing algorithms and methods.
- **New ideas**: Investigating the interface between semantic parsing and other components in the NLU pipeline, and developing new tasks and evaluation metrics for semantic parsing that solve larger NLU problems.

Below we provide a more detailed description of each chapter.

1.3.1 Chapter Overviews

In Chapters 2-4 we address a different challenge from Figure 1.15 (listed above each description), and in Chapter 5 looks look at applications of the new resources and techniques developed in Chapters 2-3.

challenge 1: Getting data?

- **Chapter 2:** We address the issue of building new datasets by looking at using **source code documentation as a resource for building parallel datasets** for semantic parsing. In doing this, we introduce 43 new automatically generated datasets, and establish a number of new baseline models on the new task of text-to-code semantic parsing.

challenge 2: Missing Data?

- **Chapter 3:** As a way of addressing the issue of missing or insufficient amounts of parallel data, we look at **learning semantic parsers from multiple datasets** using our initial source code datasets and other benchmark datasets. To do this, we develop a novel graph-based decoding framework that we use to improve on the results in Chapter 2 and to do transfer learning across different languages and domains. We also introduce new tasks such as mixed language parsing and zero-shot translation for semantic parsing.

challenge 3: Insufficient LFs?

Sub-problem	Problem Description
1. Semantic Parsing (SP)	*Translating* `input` *to* `sem`, `input` → `sem`
2. Knowledge Representation (KR)	*Defining a sufficiently expressive* `sem` *language.*
3. Reasoning (and Execution)	Going from `sem` to denotations and data
4. Semantic Parsing Resources	Finding parallel data for training semantic parsers.

	thematic content and sub-problems addressed			
Chapters and Topics	**Semantic Parsing**	**KR**	**Reasoning**	**SP Resources**
2. Semantic Parsing in Technical Docs	×			×
3. Learning from Multiple Datasets	×			×
4. Learning from Entailment	×	×	×	×
5. Applications: QA on Source Code	×	×		

Table 1.3: Thematic overview of thesis chapters (bottom) classified in terms of the full list of sub-problems (above).

- **Chapter 4:** We look at **integrating reasoning about entailment** into a data-driven semantic parsing pipeline and a new learning framework called *learning from entailment*. This new approach is motivated by cases where the logical forms being learned fail to capture basic facts about inference and entailment. We examine this issue and the effectiveness of our approach on an extended version of the benchmark Sportscaster dataset (Chen and Mooney, 2008b) annotated with entailment information and a novel recognizing textual entailment (RTE)-style evaluation.

Applications

- **Chapter 5:** We look at formalizing the source code representations being learned in Chapter 2 and 3 in terms of classical logic, and discuss some **applications of the source code datasets** investigated in Chapters 2-3, including code retrieval and API question-answering (QA). In the latter case, we describe our prototype QA system called `Function Assistant`.

While each chapter focuses primarily on the task of semantic parsing and a different variant of the resource problem, we also look at the interaction between the different sub-problems discussed in the previous section. In Table 1.3, we thematically classify each chapter according to the different sub-problems encountered.

Appendices In place of an explicit theoretical overview chapter, we include appendices to supplement the technical content covered in the chapters. This includes further descriptions of model features, theoretical questions about the underlying models, as well as other content that goes beyond the main content in each chapter. Given that existing reviews of semantic parsing (Mooney, 2008; Liang, 2016; Liang and Potts, 2015) focus exclusively on machine learning (and hence gloss over the problems of *knowledge representation* and *reasoning*), we provide a brief overview of symbolic knowledge representation and logic in Appendix C. Readers who are familiar with the technical content in each chapter can safely skip over much of the content in the appendices.

In an effort to make the technical content in each chapter self-contained, we also include additional technical details throughout each chapter that cover general machine learning and semantics concepts that are not explicitly discussed in our published work.

1.3.2 Publications

The content from these chapters is based on the following peer-reviewed publications (ordered according to chapter content):

- Richardson and Kuhn (2014)[Unixman: A Resource for Language Learning in the Unix Domain. In *Proceedings of Ninth International Conference on Language Resources and Evaluation (LREC)*. **Chapter 2**]
- Richardson and Kuhn (2017b)[Learning Semantic Correspondences in Technical Documentation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. **Chapter 2**]
- Richardson and Kuhn (2017a)[Function Assistant: A tool for NL Querying of APIs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. **Chapters 2,5**]
- Richardson et al. (2018)[Polyglot Semantic Parsing in APIs. In *Proceedings of 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. **Chapter 3**]

- Richardson and Kuhn (2016)[Learning to Make Inferences in a Semantic Parsing Task. In *Transactions of the Association for Computational Linguistics (TACL)*. **Chapter 4**]

We also use material from the following unpublished technical report:

- Richardson (2018)[A Language for Function Signature Representations. *arXiv preprint:1804.00987*. **Chapter 5**]

Resources and Reproducibility To encourage reproducibility and further experimentation, all of the data and code reported in this thesis are available through the author's GitHub account: `https://github.com/yakazimir`. This includes all code datasets reported in Chapters 2-3: `https://github.com/yakazimir/Code-Datasets` as the associated `Zubr` semantic parsing toolkit: `https://github.com/yakazimir/zubr_public` (more details in Chapter 5).

2 Semantic Parsing in Technical Documentation

2.1 NLU in Technical Documentation

2.1.1 The Idea

Technical documentation in the computer domain, such as source code documentation and other how-to manuals, provide high-level descriptions of how lower-level computer programs and utilities work. Often these descriptions are coupled with formal representations of these lower-level features, expressed in the target programming languages. For example, Figure 2.1-1 shows the source code documentation (in red/bold) for the `max` function in the Java programming language paired with the representation of this function in the underlying Java language:

```
public static long max(long a,long b)
```

This formal representation captures the name of the function, the return value, the types of arguments the function takes, among other details related to the function's place and visibility in the overall source code collection or API.

Given the high-level nature of the textual annotations, modeling the meaning of any given description is not an easy task, as it involves much more information than what is directly provided in the associated documentation. For example, capturing the meaning of the description *the greater of* might require having a background theory about quantity/numbers and relations between different quantities. A first step towards capturing the meaning, however, is learning to translate this description to symbols in the target representation, in this case to the `max` symbol. By doing this translation to a formal language, modeling and learning the subsequent semantics becomes easier since we are eliminating the ambiguity of ordinary language. Similarly, we would want to first translate the description *two long values*, which specifies the number and type of argument taken by this function, to the

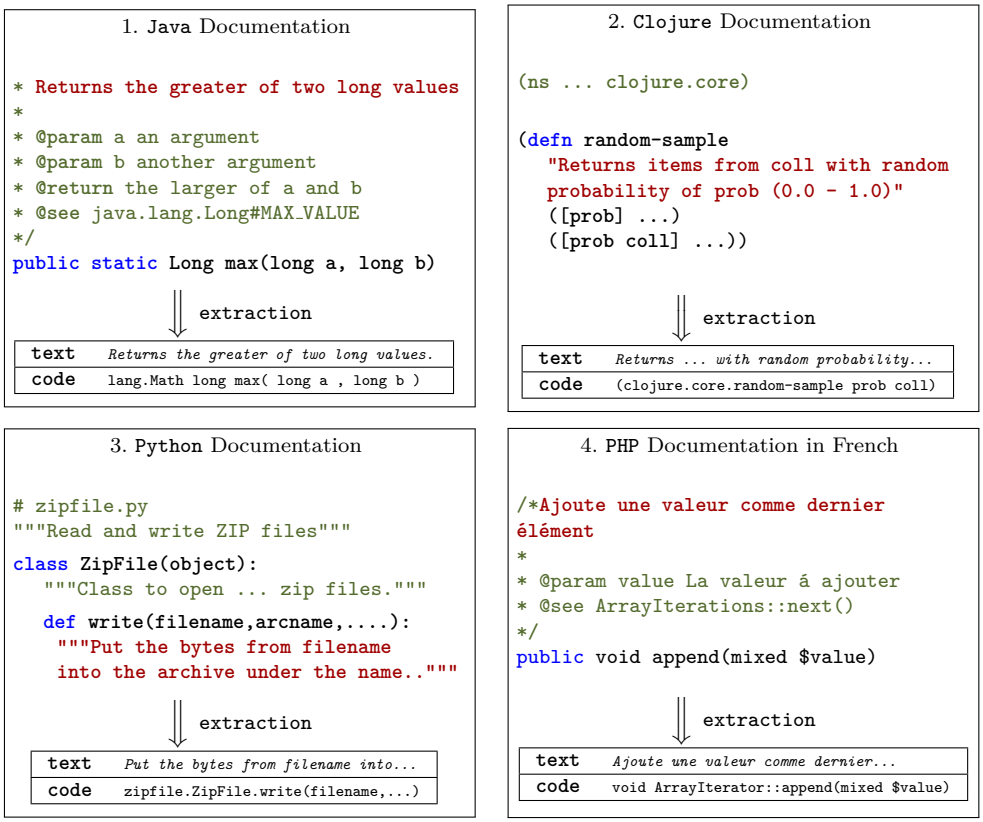


Figure 2.1: Example source code documentation across 4 programming languages and an illustration of parallel data extraction.

sequence `long a,long b`.

By focusing on translation, we can automatically create new datasets by mining these types of source code collections for sets of parallel text-representation pairs (as illustrated in Figure 2.1). Given the wide variety of available programming languages, many such datasets can be constructed, each offering new challenges related to differences in the formal representations used by different programming languages. Figure 2.1-2 shows example documentation for the Clojure programming language, which is part of the Lisp family of languages. In this case, the description *Returns random probability of* should be translated to the function name `random-sample` since it describes what the overall function does. Similarly, the argument descriptions *from coll* and *of prob* should translate to `coll` and `prob`.

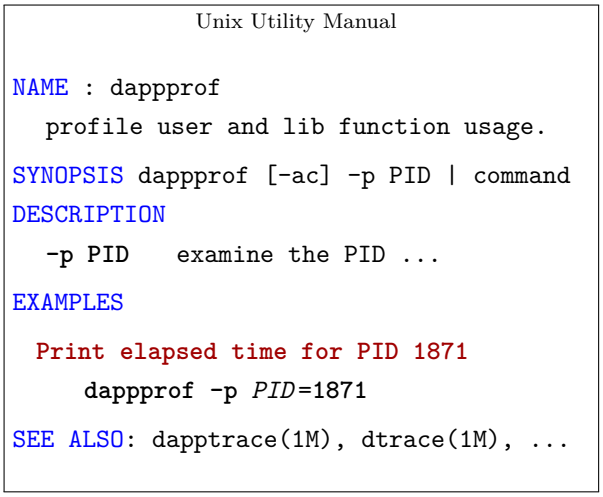


Figure 2.2: An example computer utility manual in the Unix domain. A description of an example use is shown in red.

Given the large community of programmers around the world, many source code collections are available in languages other than English. Figure 2.1-4 shows an example entry from the French version of the PHP standard library, which was translated by volunteer developers. Having multilingual data raises new challenges, and broadens the scope of investigations into this type of semantic translation.

Other types of technical documentation, such as utility manuals, exhibit similar features. Figure 2.2 shows an example manual in the domain of Unix utilities. The textual description in red/bold describes an example use of the *dappprof* utility paired with formal representations in the form of executable code. As with the previous examples, such formal representations do not capture the full meaning of the different descriptions, but serve as a convenient operationalization, or *translational semantics*, of the meaning in Unix. *Print elapsed time*, for example, roughly describes what the `dappprof` utility does, whereas *PID 1871* describes the second half of the code sequence.

In both types of technical documentation, information is not limited to raw pairs of descriptions and representations, but can include other information and clues that are useful for learning. Java function annotations include textual descriptions of individual arguments and return values (shown in green), and the Python ex-

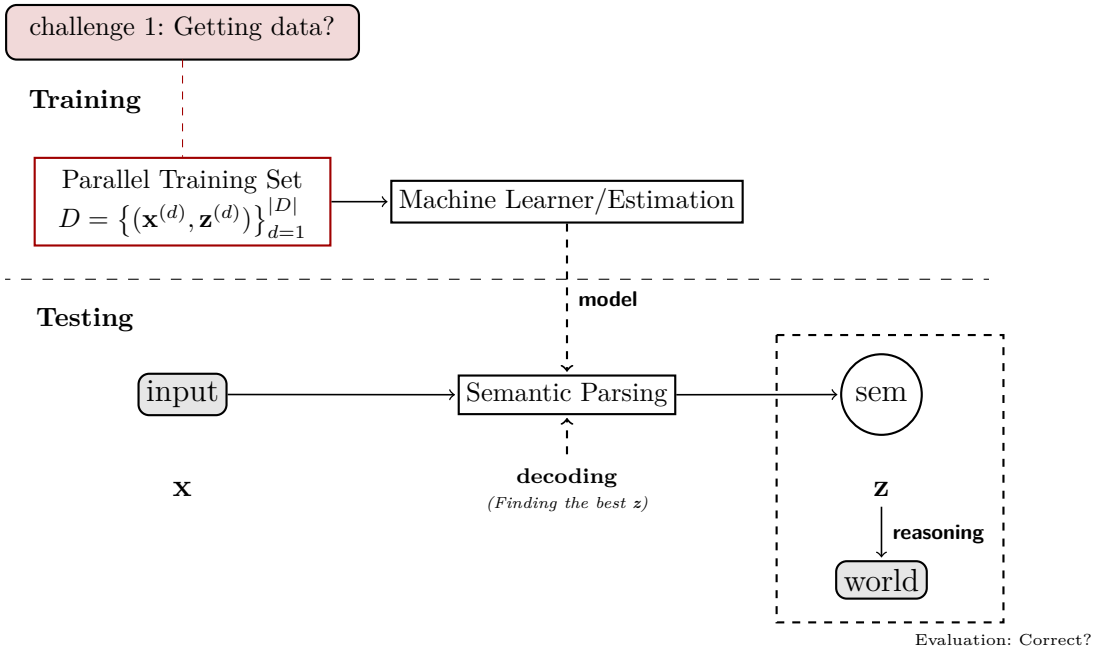


Figure 2.3: An illustration standard semantic parsing setup and the first resource challenge (from Section 1.3).

ample (in Figure 2.1.3) contains additional textual descriptions of the associated class `ZipFile`. Taxonomic information and pointers to related functions or utilities are also annotated (e.g., the `@see` section in Figure 2.1, or `SEE ALSO` section in Figure 2.2). Structural information about code sequences, and the types of abstract arguments these sequences take, are described in the `SYNOPSIS` section of the Unix manual. This last piece of information allows us to generate abstract code templates, and generalize individual arguments. For example, the raw argument `1871` in the sequence `dappprof -p 1871` can be typed as a `PID` instance, and an argument of the `-p` flag.

2.1.2 Addressing the Resource Problem

Given this type of data, a natural experiment is to see whether we can build programs that translate high-level textual descriptions (i.e., short sentence descrip-

tions) to correct formal representations (in this case, to formal function signatures). In aiming to solve this general problem, we treat the translation from text to code as a type of semantic parsing (or sequence prediction) task. More broadly, we propose using technical documentation as a general resource for building robust semantic parsing models. In doing this, we address the first resource challenge discussed in Chapter 1 (see Figure 2.3), or the problem that building parallel datasets consisting of text and formal representations usually requires considerable manual annotation effort, hence the lack of resources in the semantic parsing field. Accordingly, we can regard a source code library, or *API*, as a kind of *naturally occurring parallel corpus* consisting of text and example code representations, where, in our semantic parsing context, each code representation can be thought of as analogous to a logical form.

Summarizing the discussion above, below are some consequences of treating source code libraries as a parallel corpus and resource for semantic parsing:

1. **Automatic**: Parallel pairs of text and code are easy to extract automatically (see again Figure 2.1), obviating the need for manual annotation effort.

2. **Many datasets**: The vast quantity of programming languages and software projects makes it easy to create new semantic parsing datasets that offer unique challenges across a wide range of domains.

3. **Multilingual**: The availability of documentation in languages other than English make it possible to build multilingual parallel datasets.

4. **Background Knowledge**: Beyond raw pairs of text and code, software libraries provide lots of additional structure that can be can be used as a signal for learning.

In this chapter, we describe experiments involving 43 new software datasets that span a wide range of target programming languages and source natural languages. To our knowledge, our work is the first to look at translating source code descriptions to formal representations using such a wide variety of programming and natural languages. We also introduce several baseline models for this data that build on the language modeling and translation approach of Deng and Chrupała

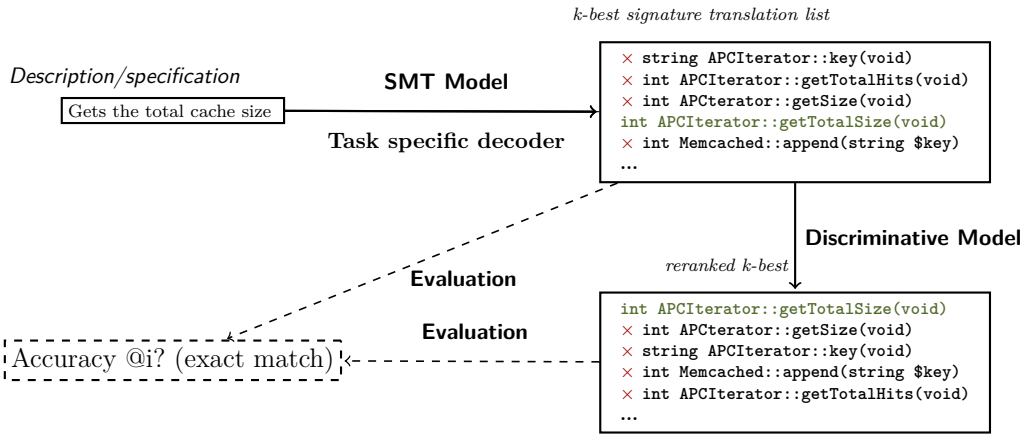


Figure 2.4: Our main code translation model and our evaluation at test time.

(2014). Technically, our main approach, as illustrated in Figure 2.4, has the following two components: a simple SMT model and custom decoder (described in Section 3.2.1 alongside other types of baseline *language models*) and discriminative reranking model (described in Section 3.2.1) that, in part, exploits additional features about the global API.

In doing these experiments, we aim to address the following research questions:

- **Translation Difficulty**: How hard is the text-to-code translation problem (when treated as a semantic parsing or translation problem) using these datasets, and what types of models work/do not work?
- **Domain Information**: Can background knowledge about the target source code library be used to improve the translation?

In general, our experiments (discussed in Section 2.4-3.5) show that simple SMT baseline models perform the best, and that modest improvements can be achieved using a conventional discriminative model (Zettlemoyer and Collins, 2009) that, in part, exploits document-level features from the underlying software libraries. (Given our ultimate interest in doing NLU in the source code domain, we later describe in Chapter 5 ways of formalizing the source code representations being learned in this chapter in terms of classical logic).

2.2 Related Work

Our work is situated within research on semantic parsing (SP), which focuses on the problem of generating formal meaning representations from text for natural language understanding applications (for a full overview, see Chapter 1). This chapter focuses primarily on new resources for SP, since, to date, most benchmark datasets are limited to small controlled domains, such as geography and navigation. While attempts have been made to do open-domain SP using larger, more complex datasets (Berant et al., 2013; Pasupat and Liang, 2015), such resources are still scarce. In Figure 2.1, we compare the details of one widely used dataset, GeoQuery (Zelle and Mooney, 1996), to our new datasets. Our new resources are on average much larger than GeoQuery in terms of the number of example pairs, and the size of the different language vocabularies. Most existing datasets are also primarily English-based, while we focus on learning in a multilingual setting using several new moderately sized datasets.

Within SP, there has also been work on situated or grounded learning, that involves learning in domains with weak supervision and indirect cues (Liang, 2016; Richardson and Kuhn, 2016). This has sometimes involved learning from automatically generated parallel data and representations (Chen and Mooney, 2008a) of the type we consider in this chapter. Here one can find work in technical domains, including learning to generate regular expressions (Manshadi et al., 2013; Kushman and Barzilay, 2013) and other types of source code (Quirk et al., 2015), which ultimately aim to solve the problem of natural language programming. We view our work as one small step in this general direction.

Our work is also related to software components retrieval and builds on the approach of Deng and Chrupała (2014). Robustly learning the translation from language to code representations can help to facilitate natural language querying of API collections (Lv et al., 2015). As part of this effort, recent work in machine learning has focused on the similar problem of learning code representations using resources such as StackOverflow and Github. These studies primarily focus on learning longer programs (Allamanis et al., 2015) as opposed to function representations, or focus narrowly on a single programming language such as Java (Gu et al., 2016) or on related tasks such as text generation (Iyer et al., 2016; Oda

Dataset	#Pairs	#Descr. Symbols	#Words	Vocab.	Example text-code pairs ($\mathbf{x}, \mathbf{z}$)
Java	7,183	4,804	4,072	82,696	3,721	**x** : Compares this Calendar to the specified Object. **z** : `boolean util.Calendar.equals(Object obj)`
Ruby	6,885	1,849	3,803	67,274	5,131	**x** : Computes the arc tangent given y and x. **z** : `Math.atan2(y,x)` → `Float`
PHP_{en}	6,611	13,943	8,308	68,921	4,874	**x** : Delete an entry in the archive using its name. **z** : `bool ZipArchive::deleteName(string $name)`
Python	3,085	429	3,991	27,012	2,768	**x** : Remove the specific filter from this handler. **z** : `logging.Filterer.removeFilter(filter)`
Elisp	2,089	1,365	1,883	30,248	2,644	**x** : This returns the total height of the window. **z** : `(window-total-height window round)`
Haskell	1,633	255	1,604	19,242	2,192	**x** : Extract the second component of a pair. **z** : `Data.Tuple.snd :: (a, b) -> b`
Clojure	1,739	–	2,569	17,568	2,233	**x** : Returns a lazy seq of every nth item in coll. **z** : `(core.take-nth n coll)`
C	1,436	1,478	1,452	12,811	1,835	**x** : Returns the current file position of the stream. **z** : `long int ftell(FILE *stream)`
Scheme	1,301	376	1,343	15,574	1,756	**x** : Returns a new port with type port-type and state. **z** : `(make-port port-type state)`
Unix	921	940	1,000	11,100	2,025	**x** : To get policies for a specific user account. **z** : `pwpolicy -u username -getpolicy`
GeoQuery	880	–	167	6,663	279	**x** : What is the tallest mountain in America? **z** : `(highest(mountain(loc_2(countryid usa))))`

Table 2.1: Description of our English Stdlib corpus collection (with example signatures shown in a conventionalized format)

et al., 2015). To our knowledge, none of this work has been applied to languages other than English or such a wide variety of programming languages.

2.3 Text-to-Component Translation: Problem Description

We use the term *technical documentation* to refer to two types of resources: textual descriptions inside of *source code collections* and *computer utility manuals*. The first type includes high-level descriptions of functions in source code. The second type includes a collection of Unix manuals, also known as man pages. Both types

include pairs of brief sentence descriptions and code representations.

We will refer to the target representations in these resources as *API components*, or components. In source code, components are formal representations of functions, or *function signatures* (Deng and Chrupała, 2014). The form of a function signature varies depending on the resource, but in general gives a specification of how a function is named and structured. The example function signatures in Figure 2.1. for example, all specify a function name, a list of arguments, and other optional information such as a return value and a namespace. Components in utility manuals are short executable code sequences intended to show an example use of a utility. We assume typed code sequences following Richardson and Kuhn (2014), where the constituent parts of the sequences are abstracted by type.

Given a training set of example text-component pairs, $D = \{(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})\}_{d=1}^{|D|}$, the goal is to learn how to generate correct, well-formed components $\mathbf{z} \in \mathcal{C}$ for each input $\mathbf{x}$, or some function (or weighted relation) $\mathtt{sp}$:

$$\mathtt{sp} : \mathbf{x} \rightarrow \mathbf{z}$$

Viewed as a SP problem, this treats the target components as a kind of formal meaning representation, analogous to a logical form. In our experiments, we assume that the complete set of output components are known. In the API documentation sets, this is because each source library contains a finite number of defined function representations, roughly corresponding to the number of pairs as shown in Figure 2.1. For a given input, therefore, the goal is to find the best candidate function translation within the space of the total API components $\mathcal{C}$ (Deng and Chrupała, 2014) (see Figure 2.4).

Given these constraints, our setup closely resembles that of Kushman et al. (2014), who learn to parse algebra word problems using a small set of equation templates. Their approach is inspired by template-based information extraction, where templates are recognized and instantiated by slot-filling. Our function signatures and code templates have a similar slot-like structure, consisting of slots such as return value, arguments, function name and namespace (see 3.6 for a formalization of the signature representations).

2.3.1 Language Modeling Baseline Formulation

Existing approaches to SP formalize the mapping from language to formal languages using a variety of formalisms including CFGs (Börschinger et al., 2011), CCGs (Kwiatkowski et al., 2010), synchronous CFGs (Wong and Mooney, 2007) (see review in Chapter 1). Deciding to use one formalism over another is often motivated by the complexities of the target representations being learned. For example, recent interest in learning graph-based representations such as those in the AMR bank (Banarescu et al., 2013) requires parsing models that can generate complex graph shaped derivations such as CCGs (Artzi et al., 2015) or HRGs (Peng et al., 2015). Given the simplicity of our API representations, we opt for a simple SP model that exploits the finiteness of our target representations.

We formulate the problem of component translation as a general language modeling problem following Deng and Chrupała (2014) (henceforth DC). For a given *query* sequence or text $\mathbf{x} = (x_1, .., x_{|\mathbf{x}|})$ and *component* sequence $\mathbf{z} = (z_1, .., z_{|\mathbf{z}|})$, the probability of the component given the query is defined as follows:

$$p(\mathbf{z} \mid \mathbf{x}) \propto p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z}) \tag{2.1}$$

By ignoring the denominator and assuming a uniform prior over the probability of each *valid* component $\mathbf{z} \in \mathcal{C}$, the problem reduces to computing $p(\mathbf{x} \mid \mathbf{z})$, which is where language modeling is used (Zhai and Lafferty, 2004). Given each word x_i in the query, a unigram model is defined as:

$$p(\mathbf{x} \mid \mathbf{z}) = \prod_{i=1}^{|\mathbf{x}|} p(x_i \mid \mathbf{z}). \tag{2.2}$$

Using this formulation, we can then define different models to estimate $p(x_j \mid \mathbf{z})$ (i.e., the probability of each word x_j in $\mathbf{x}$ given a candidate signature $\mathbf{z}$).

Term Matching As a baseline for $p(x_i \mid \mathbf{z})$, DC define a *term matching* approach that exploits the fact that many queries in our English datasets share vocabulary with the target component vocabulary. A smoothed version of this baseline is defined below, where $f(x_i \mid \mathbf{z})$ is the frequency of matching terms in the target

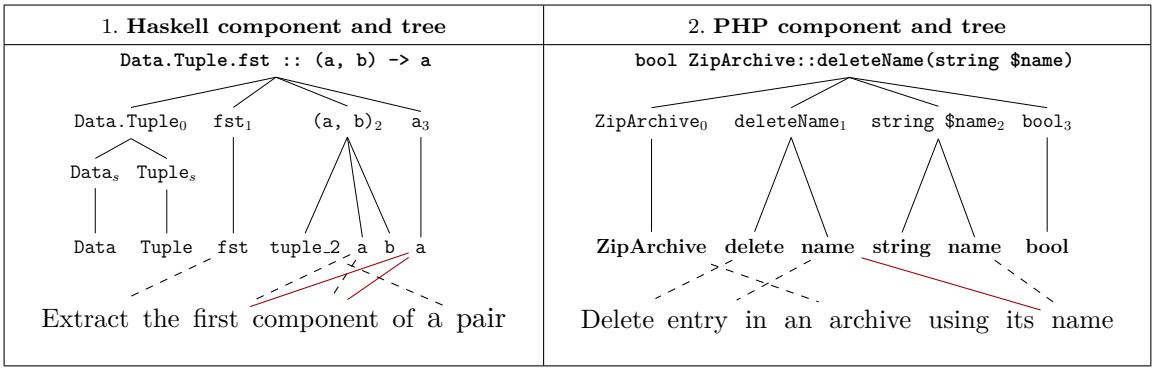


Figure 2.5: Example tokenized components with tree representations and alignments with text.

signature, $f(x_i \mid \mathcal{C})$ is frequency of the term word in the overall documentation collection, and λ is a smoothing parameter (for Jelinek-Mercer smoothing):

$$p(\mathbf{x} \mid \mathbf{z}) = \prod_{x=1}^{|\mathbf{x}|} \Big[(1-\lambda) f(x_i \mid \mathbf{z}) + \lambda f(x_i \mid \mathcal{C}) \Big] \tag{2.3}$$

Translation Model In order to account for the co-occurrence between non-matching words and component terms, DC employ a word-based translation model, which models the relation between natural language words x_i and individual component terms z_j. In this chapter, we limit ourselves to sequence-based word alignment models (Och and Ney, 2003), which factor in the following manner:

$$p(\mathbf{x} \mid \mathbf{z}) = \prod_{i=1}^{|\mathbf{x}|} \sum_{j=0}^{|\mathbf{z}|} \Big[p_t(x_i \mid z_j) p_d(j \mid i, \mathbf{x}, \mathbf{z}) \Big] \tag{2.4}$$

Here each $p_t(x_i \mid z_j)$ defines an (unsmoothed) multinomial distribution over a given component term z_j for all words x_j. The function p_d includes additional parameters about the relative length of each alignment and the size of the input and output. This function assumes different forms according to the particular alignment model being used. We consider three different types of alignment models each defined in

the following way:

$$p_d(j \mid i, \mathbf{x}, \mathbf{z}) = \begin{cases} \frac{1}{|\mathbf{z}|+1} & \text{1. IBM Model 1} \\ a(j \mid i, |\mathbf{x}|, |\mathbf{z}|) & \text{2. IBM Model 2} \\ a(t(j) \mid i, |\mathbf{x}|, tlen(\mathbf{z})) & \text{3. Tree Model} \end{cases} \tag{2.5}$$

Models 1-2 are the classic IBM word-alignment models of Brown et al. (1993). IBM Model 1 assumes a uniform distribution over all positions, and is the main model investigated in DC. For comparison, we also experiment with IBM Model 2, where each j is the string position of the alignment in the component input, and $a(..)$ defines a multinomial distribution such that $\sum_{j=0}^{|\mathbf{z}|} a(j \mid i, |\mathbf{x}|, |\mathbf{z}|) = 1.0$.

We also define a new tree based alignment model (3) that takes into account the syntax associated with the function component representations. Each $t(j)$ is the relative tree position of the alignment point and $tlen(\mathbf{z})$ is the length of the tree associated with $\mathbf{z}$. This approach assumes a tree representation for each $\mathbf{z}$. We generated these trees heuristically by preserving the information that is lost when components are converted to a linear sequence representation. Two example trees are shown in Figure 2.5 for Haskell and PHP, where the red solid lines indicate the types of potential alignment errors that this model aims to avoid.

Learning and the EM Algorithm

For our translation models, learning is done by applying the standard training procedure of Brown et al. (1993). This algorithm is based on the expectation maximization (EM) algorithm (Dempster et al., 1977) (see also Neal and Hinton (1998); Bishop (2006)), which is an iterative maximum likelihood estimation technique designed for estimating models with hidden variables. Since the EM algorithm is encountered in different parts of this thesis, we describe the basic theory behind the algorithm using our translation models as an example.

Word-based translation models (of the type considered above) compute the conditional probability in Equation 2.4 by summing over the probability of all many-

Algorithm 1 Expectation-Maximization for IBM Model 1

Input: Parallel corpus $D = \{(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})\}_{d=1}^{|D|}$, for each $\mathbf{x} = x_1, .., x_{|\mathbf{x}|}$ with each $x_j \in \Sigma_{\texttt{nl}}$ and $\mathbf{z} = z_0 = nil, z_1, .., z_{|\mathbf{z}|}$ with each $z_j \in \Sigma_{\texttt{sem}}$
Output: lexical probability function p_t

$p_{\theta^0} \leftarrow 0.0, t \leftarrow 0$ ▷ Initialize parameters at initial step 0
repeat
  $c(x \mid z) \leftarrow 0, \forall x \in \Sigma_{\texttt{nl}}, \forall z \in \Sigma_{\texttt{sem}}$ ▷ Initialize counters to collect expected counts
  $b(z) \leftarrow 0, \forall z \in \Sigma_{\texttt{sem}}$
  for $(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})$ from $d = 1$ up to $\mid D \mid$ **do** ▷ Start of E-Step: evaluate $p_{\theta^t}(a \mid \mathbf{x}, \mathbf{z})$
    for i from 1 up to $\mid \mathbf{x}^{(d)} \mid$ **do** ▷ Compute normalization n
      $n_i \leftarrow \sum_j^{|\mathbf{z}^{(d)}|} p_{\theta^t}(x_i^{(d)} \mid z_j^{(d)})$
    for i from 1 up to $\mid \mathbf{x}^{(d)} \mid$ **do**
      for j from 0 up to $\mid \mathbf{z}^{(d)} \mid$ **do**
        $c(x_i^{(d)} \mid z_j^{(d)}) \leftarrow c(x_i^{(d)} \mid z_j^{(d)}) + p_{\theta^t}(x_i^{(d)} \mid z_j^{(d)})/n_i$ ▷ Update expect. counts
        $b(z_j^{(d)}) \leftarrow c(x_i^{(d)} \mid z_j^{(d)}) + p_{\theta^t}(x_i^{(d)} \mid z_j^{(d)})/n_i$
  for all $z \in \Sigma_{\texttt{sem}}$ **do** ▷ The M-Step, find $\theta^{t+1} = \arg\max_\theta \mathcal{Q}(\theta \mid \theta_t)$
    for all $x \in \Sigma_{\texttt{nl}}$ **do**
      $p_{\theta^{t+1}}(x \mid z) \leftarrow c(x \mid z)/b(z)$
  $t \leftarrow t + 1$
until converged
return p_{θ^t}

to-one word alignments from $\mathbf{x} \rightarrow \mathbf{z}$ (denoted as $\mathcal{A}$):

$$p(\mathbf{x} \mid \mathbf{z}) = \sum_{a \in \mathcal{A}} p(\mathbf{x}, a \mid \mathbf{z}) \tag{2.6}$$

where $|\mathcal{A}|$ is equal to $(|\mathbf{z}| + 1^{|\mathbf{x}|})$. Given such a large space, doing this computation for some word-based models is intractable. In the case of IBM Models 1-2 and our tree model, however, such a computation can be done using Equations 2.6-2.7 (via some algebraic manipulation not shown) in time $O(|\mathbf{z}||\mathbf{x}|)$.

The goal of learning is to find the set of parameters θ that maximize the (log)

likelihood of our training data D, expressed as follows:

$$\theta^* = \max_{\theta} \log \prod_{d=1}^{|D|} \left[p_\theta(\mathbf{x}^{(d)} \mid \mathbf{z}^{(d)}) \right] \tag{2.7}$$

$$= \max_{\theta} \sum_{d=1}^{|D|} \log \left[\sum_{a \in \mathcal{A}} p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \right] \tag{2.8}$$

The difficulty in finding such a model is that the alignments a that drive the translation process are not directly observed in D, and hence are latent variables. The general EM algorithm breaks the problem into two steps. The first step, or E-step, involves finding the *expected value* of the latent variables (i.e., the different alignments a) using an auxiliary *posterior distribution* $p_{\theta^t}(a \mid \mathbf{x}, \mathbf{z})$ defined over some (possibly random) parameters θ^t. The idea is to then use these expected values and initial parameters (which provide a lower bound on the likelihood function) to compute the *complete data* likelihood in Equation 2.10 using an additional set of parameters θ, as shown below:

$$\mathcal{Q}(\theta \mid \theta^t) = \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} \left[p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \right] \tag{2.9}$$

$$= \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} \left[\left[\frac{p_{\theta^t}(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)})}{p_{\theta^t}(\mathbf{x}^{(d)} \mid \mathbf{z}^{(d)})} \right] \log p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \right] \tag{2.10}$$

The second step, or M-Step, then involves finding in particular the parameters θ^{t+1} that maximize Equation 2.11:

$$\theta^{t+1} = \arg\max_{\theta} \mathcal{Q}(\theta \mid \theta^t) \tag{2.11}$$

After the M-step, the algorithm returns to the E-step and the overall process repeats until a convergence point is reached.

We note that Equations 2.11-2.13 only show the general form of the EM algorithm for our translation model, and the particular computation required for correctly collecting the expected counts and performing updates requires solving the general arg max in Equation 2.11 for each individual model. Recalling that alignments rely on individual word translation probabilities, computing the esti-

Algorithm 2 Linear Rank Decoder

Input: Input $\mathbf{x}$, Components $\mathcal{C}$, rank k, model $\mathcal{M}$, sort function K-BEST
Output: Top k components ranked by $\mathcal{A}$ model score p

```
1: SCORES ← [ ]                                  ▷ Initialize list of scored components
2: for each component z ∈ C do
3:     p ← SCORE_M(x, c)                         ▷ Score pair (x, c) using M
4:     SCORES ← SCORES + (z, p)                  ▷ Add component and score to candidate list
5: return K-Best(SCORES,k)                       ▷ Return the k best components
```

mation involves counting the expectation of individual word pair occurrences (using a count function c) as shown below (Och and Ney (2003), for a full derivation of this counting technique from the EM objective, see Brown et al. (1993)):

$$c(x \mid z; D) = \sum_{d=1}^{|D|} \Big[\sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \sum_{j}^{|\mathbf{x}^{(d)}|} \delta(x, x_j^{(d)}) \delta(z, z_{a_{(j)}}^{(d)}) \Big]$$

Algorithm 1 shows the full EM algorithm for IBM Model 1. By applying Equations 2.6 and 2.7, the E-step in this case takes the following form (as shown in line 11 of Algorithm 1):

$$c(x \mid z; D) = \sum_{d=1}^{|D|} \Big[\frac{p_{\theta^t}(x \mid z)}{\sum_{i}^{|\mathbf{z}^{(d)}|} p_{\theta^t}(x \mid z_i)} \sum_{j=1}^{|\mathbf{x}^{(d)}|} \delta(x, x_j^{(d)}) \sum_{i=0}^{|\mathbf{z}^{(d)}|} \delta(z, z_i^{(d)}) \Big] \tag{2.12}$$

Then p_{θ^t} is re-estimated to find $p_{\theta^{t+1}}$ as follows (lines 16-20 in Algorithm 1):

$$p_{\theta^{t+1}}(x \mid z) = \frac{c(x \mid z; D)}{\sum_{x'} c(x' \mid z; D)} \tag{2.13}$$

While the general form of $\mathcal{Q}$ might seem arbitrary at first glance, it comes with the guarantee that each iteration (up to a convergence point) will produce a new set of parameters that increase the likelihood of the training data D (see Appendix A for more details).

Ranking and Constrained Decoding

Algorithm 2 shows how to rank API components. For a text input $\mathbf{x}$, we iterate through all known API components $\mathcal{C}$ and assign a score using a model $\mathcal{M}$. We then rank the components by their scores using a K-Best function. For our translation models, this method serves as a type of word-based decoding algorithm, which in part solves the following decoding problem (i.e., the problem of finding the most likely output $\mathbf{z}^*$ given an input $\mathbf{x}$):

$$\mathbf{z}^* = \arg\max_{\mathbf{z}} \Big\{ p(\mathbf{z} \mid \mathbf{x}) \Big\} \tag{2.14}$$

$$= \arg\max_{\mathbf{z}} \Big\{ \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{p(\mathbf{x})} \Big\} \qquad \text{Bayes' Theorem} \tag{2.15}$$

$$= \arg\max_{\mathbf{z}} \Big\{ p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z}) \Big\} \qquad \text{constant p}(\mathbf{x}) \tag{2.16}$$

Even given our simplified model with a uniform prior, this problem is known to be intractable in the general case for the translation models under consideration (Knight, 1999). Our approach, however, exploits the the finiteness of the target prediction space, and is designed to do k-best decoding in order to accommodate reranking and our evaluation method introduced below. The complexity of the scoring procedure (lines 2-4) is linear over the number components $|\mathcal{C}|$ in the target library. In practice, we implement the K-Best sorting function on line 5 as a binary insertion sort, resulting in an overall complexity of $O(|\mathcal{C}| \log |\mathcal{C}|)$ (we note that this can be improved to linear time by using a min-heap or the order statistic algorithm (Blum et al., 1973) in place of our insertion sort).

While iterating over $|\mathcal{C}|$ components might not be feasible given large APIs or more complicated formal languages with compositionality, a more clever decoding algorithm could be applied, e.g., one based on the lattice decoding approach of Dyer et al. (2008). We consider a more efficient decoder model of this type (i.e., one that does not bind the complexity of the search to the size of out the output space) in the next chapter, but nonetheless use the above method to establish initial baseline results and note that for the datasets and experiments under consideration in this chapter, we did not encounter efficiency issues since our APIs on average contain only a few thousand functions.

2.3.2 Discriminative Approach

In this section, we introduce a more informed model that aims to improve on the previous baseline methods. While the previous models are restricted to word-level information, we extend this approach by using a discriminative reranking model over our translation model output that captures phrase information to see if this leads to an improvement. This model can also capture document-level information from the APIs, such as the additional textual descriptions of parameters, *see also* declarations or classes of related functions and syntax information.

Modeling

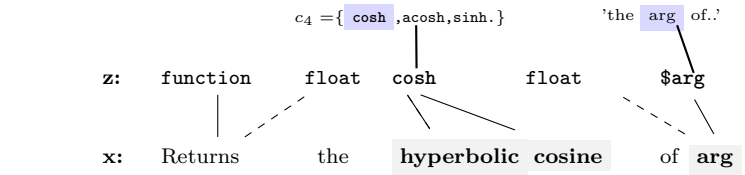
Building on previous work in SP (Zettlemoyer and Collins, 2009; Liang et al., 2011), our model is defined as a conditional log-linear model over components $\mathbf{z} \in \mathcal{C}$ (akin to *derivations* in a traditional grammar-based semantic parser) with parameters $\theta \in \mathbb{R}^b$, and a set of feature functions $\phi(\mathbf{x}, \mathbf{z})$:

$$p_\theta(\mathbf{z} \mid \mathbf{x}) = \frac{e^{\theta \cdot \phi(\mathbf{x},\mathbf{z})}}{\sum_{\mathbf{z'}} e^{\theta \cdot \phi(\mathbf{x},\mathbf{z'})}} \tag{2.17}$$

where each $\theta \cdot \phi(\mathbf{x}, \mathbf{z})$ is defined as follows (for individual feature weights θ_j and feature values ϕ_j):

$$\theta \cdot \phi(\mathbf{x}, \mathbf{z}) = \sum_{i=1}^{b} \theta_i \phi_i(\mathbf{x}, \mathbf{z}) \tag{2.18}$$

Formally, our training objective takes the same form as Equation 2.9, with the important difference that our log-linear models do not have the same latent variables a as our translation models. One of the technical difficulties in optimizing objectives like the one in Equation 2.10 is that the log gets stuck outside the sum over the latent variables, which makes it hard for the log to decompose the rest of the likelihood term (Daumé III, 2012). Without these latent variables, such as in the case above, we can instead decompose the likelihood directly and use stochastic gradient ascent (LeCun et al., 1998) to optimize our objective, as described later in this section.



ϕ(x,z) =

Model score:	is it in top 5..10?
Pairs/Alignments:	(hyperbolic, `cosh`) = 1, (cosine, `cosh`) = 1, ...
Phrases:	(hyperbolic cosine, `cosh`) = 1, (of arg, `float $arg`) = ...
See also:	(hyperbolic, c_4 = {`cos,..`}) = 1, (arg, c_4) = 1, ...
In Descr.:	(arg, , `$arg`) = 1, (arg , `float`) = 0, ...
Trees/Matches	(hyperbolic, `cosh`, NAME_NODE) = 1, number of matches= ...

Figure 2.6: An illustration of the features used for our reranker.

Features

Our model uses word-level features, such as word match, word pairs, as well as information from the underlying aligner model such as Viterbi alignment information and model score. Two additional categories of non-word features are described below. An illustration of the feature extraction procedure is shown in Figure 2.6 and a full list of features with some analysis is included in Appendix A.

Phrase Features We extract phrase features (e.g., (hyper. cosine, `cosh`) in Figure 2.6) from example text component pairs by applying standard word-level heuristics on top of our word alignment models. As standardly done in phrase-based translation, we derive many-to-many alignments between each $\mathbf{x}$ and $\mathbf{z}$ by using symmetric translation models trained in both directions (i.e., $\mathbf{x} \rightarrow \mathbf{z}$, as above, and $\mathbf{z} \rightarrow \mathbf{x}$) (Koehn et al., 2003). With these many-to-many alignments, phrase extraction works by finding all phrase pairs $(\mathbf{x}_i^j, \mathbf{z}_{i'}^{j'})$ (where i, i', j, j' indicate the start and end spans in $\mathbf{x}, \mathbf{z}$) that are *consistent* with the underlying alignment

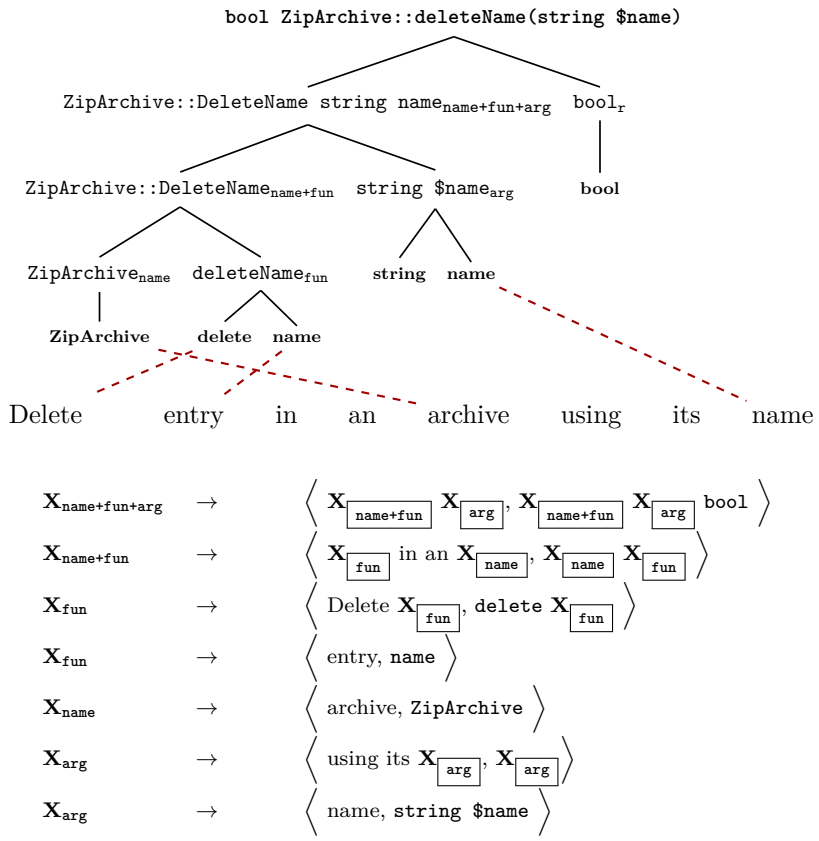


Figure 2.7: An example hierarchical phrase-based grammar extracted from an aligned component tree and text pair (top).

A, where consistency is defined in the following way (Koehn, 2009):

$$
\begin{aligned}
&(\mathbf{x}_i^j, \mathbf{z}_{i'}^{j'}) \text{ is consistent with } A \\
&\Leftrightarrow \\
&\forall x_i \text{ in } \mathbf{x}_i^j : (x_i, z_{i'}) \in A \Rightarrow z_{i'} \text{ in } \mathbf{z}_{i'}^{j'} \\
&\wedge\ \forall z_{i'} \text{ in } \mathbf{z}_{i'}^{j'} : (x_i, z_{i'}) \in A \Rightarrow x_{i'} \text{ in } \mathbf{x}_i^j \\
&\wedge\ \exists x_i \text{ in } \mathbf{x}_i^j, z_{i'} \text{ in } \mathbf{z}_{i'}^{j'} : (x_i, z_{i'}) \in A
\end{aligned}
$$

Additional features, such as phrase match/overlap, tree positions of phrases (see again Figure 2.5), are defined over the extracted phrases.

We also extract hierarchical phrase (Chiang, 2007) features using a variant of the SAMT method of Zollmann and Venugopal (2006) and the component syntax trees. In general, hierarchical phrase-based translation extends ordinary phrase-based translation by formalizing phrase rules in terms of synchronous rewrite rules (inspired by syntax-directed translation in compiler design (Aho and Ullman, 1969)) that take the following form:

$$X \rightarrow \left\langle \psi, \rho, \sim \right\rangle$$

where X is a non-terminal in a given grammar, ψ, ρ are sequences of words and non-terminals from $\mathbf{x}$ and $\mathbf{z}$ (respectively), and $\sim$ is some one-to-one correspondence between the non-terminals in ψ and ρ. Accordingly, for each *consistent* phrase pair $(\mathbf{x}_i^j, \mathbf{z}_{i'}^{j'})$ where $\mathbf{z}_{i'}^{j'}$ corresponds to a subtree in the component tree with label l, the following is an initial rule:

$$X_l \rightarrow \left\langle \mathbf{x}_i^j, \mathbf{z}_{i'}^{j'} \right\rangle$$

More complex rules are then constructed by replacing sub-phrases in these initial rules with non-terminals from rules elsewhere in our rule set. For example, assuming a rule $X_l \rightarrow \left\langle \psi, \rho \right\rangle$ and a phrase pair $(\mathbf{x}_i^j, \mathbf{z}_{i'}^{j'})$ with label l such that $\psi = \psi_1 \mathbf{x}_i^j$ and $\rho = \rho_1 \mathbf{z}_{i'}^{j'} \rho_2$, we create a new unary expansion rule:

$$X_l \rightarrow \left\langle \psi_1 X_{\boxed{l}} \psi_2, \rho_1 X_{\boxed{l}} \rho_2 \right\rangle$$

The same procedure is also used for finding binary rule patterns (i.e., cases in which ψ and ρ contain two non-terminal patterns) that are consistent with a given glue grammar. An binarized version of the component tree in Figure 2.5-2 and description pair is show in Figure 2.7, along with an example hierarchical phrase grammar. Glue rules in this case dictate that `fun` constituents (e.g., `deleteName`) can be combined with namespace, or `name` constituents (e.g., `ZipArchive`), and that the resulting `name+fun` constituent can be combined with argument constituents `arg` (e.g., `string $name`) Given a text and candidate component pair, we use a CKY-style chart filling procedure (see Algorithm 8) to find candidate rules subject to the underlying alignment and these glue rules.

Algorithm 3 Online Learner for Reranking

Input: Dataset D, components $\mathcal{C}$, iterations T, rank k, learning rate α, model $\mathcal{A}$, ranker function RANK
Output: Weight vector θ

$\theta \leftarrow 0$ ▷ Initialize the weight vector
for $t \in 1..T$ **do**
  for pairs $(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})$ for $d \in 1...|D|$ **do**
    $\mathcal{S} \leftarrow \textsc{Rank}(\mathbf{x}^{(d)}, \mathcal{C}, k, \mathcal{A})$ ▷ Score and rank candidate components
    $\Delta \leftarrow \phi(\mathbf{x}^{(d)}, \mathbf{z}^{(d)}) - \mathbb{E}_{\mathbf{z'} \in \mathcal{S} \cup \{\mathbf{z}^{(d)}\} \sim p(\mathbf{z'}|\mathbf{x}^{(d)};\theta)} \left[\phi(\mathbf{x}^{(d)}, \mathbf{z'})\right]$ ▷ Compute gradients
    $\theta \leftarrow \theta + \alpha\Delta$ ▷ Perform an online update
return θ

Document Level Features Document features are of two categories. The first includes additional textual descriptions of parameters, return values, and modules. One class of features is whether certain words under consideration appear in the `@param` and `@return` descriptions of the target components. For example, the *arg* token in Figure 2.6 appears in the textual description of the `$arg` parameter elsewhere in the documentation string.

Other features relate to general information about abstract symbol categories, as specified in *see-also* assertions, or *hyper-link* pointers. By exploiting this information, we extract general classes of functions, for example the set of hyperbolic functions (e.g., `sinh`, `cosh`, shown as c_4 in Figure 2.6), and associate these classes with words and phrases (e.g., *hyperbolic* and *hyperbolic cosine*, see Appendix A for more details).

Learning

To optimize our objective, we use Algorithm 3, which is a variant of stochastic gradient ascent (or descent, SGD). In the general case, SGD works according to the following online update rule applied to each data point d in our dataset D:

$$\theta \leftarrow \theta + \alpha\Big(\nabla \mathcal{O}^{(d)}(\theta)\Big) \tag{2.19}$$

where $\theta \in \mathbb{R}^b$ are our parameters, α is step size or learning rate parameter, $\mathcal{O}$ is our objective, and ∇ is a gradient, or vector of partial derivatives $[\frac{\partial \mathcal{O}^{(d)}}{\partial \theta_1}, ..., \frac{\partial \mathcal{O}^{(d)}}{\partial \theta_b}]$. We

estimate the model parameters θ using a k-best approximation (using the k-best list $\mathcal{S}$ computed at line 4) of the standard SGD updates (lines 5-6). Accordingly, differentiating the objective function (as shown in line 5) with respect to each individual feature θ_j and pair $(\mathbf{x}, \mathbf{z})$ leads to the following:

$$\frac{\partial \log p_\theta(\mathbf{z} \mid \mathbf{x})}{\partial_{\theta_j}} = \phi_j(\mathbf{x}, \mathbf{z}) - \sum_{\mathbf{z'} \in \mathcal{S} \cup \{\mathbf{z}\}} \Big[\phi_j(\mathbf{x}, \mathbf{z'}) p_\theta(\mathbf{z'} \mid \mathbf{x}) \Big] \tag{2.20}$$

We note that while we use the ranker described in Algorithm 2, any suitable ranker or decoding method could be used here.

2.4 Experimental Setup

We experiment with a total of 43 API datasets drawn from the Stdlib dataset (detailed in Table 2.1 and Table 2.3, from Richardson and Kuhn (2014, 2017b)) and the Py27 dataset (detailed in Table 2.2, from Richardson and Kuhn (2017a)). In each case, datasets are quantified in terms of number of total sentence/signature pairs (# **Pairs**), number of target symbols (# **Symbols**), number of natural language tokens (# **Words**) and natural language vocabulary size (# **Vocab**). In the Stdlib case, we also include information about the number of additional textual descriptions provided (# **Descr.**). In this section, we describe these different datasets and our general experimental setup that follows Deng and Chrupała (2014); Richardson and Kuhn (2017b,a).

As discussed above, in Table 2.1 we compare against one benchmark SP dataset called GeoQuery, which is on average considerably smaller than most of our datasets (both in terms of number of pairs as well as scope and content). One motivation for this work is to provide a larger set of resources for SP development in order to attack the *resource problem for SP* discussed at the onset. In the next chapter, we provide a closer comparison of results on GeoQuery and our code datasets.

2.4.1 Datasets

Stdlib Dataset Our Stdlib collection consists of the standard library for nine programming languages, which are listed in Table 2.1. We also use the translated

version of the PHP collection for six additional languages, the details of which are shown in Table 2.3. The Java dataset was first used in DC, while we extracted all other datasets for this work. In addition, we use a collection of man pages from Richardson and Kuhn (2014) that includes 921 text-code pairs that span 330 Unix utilities and man pages. Using information from the synopsis and parameter declarations, the target code representations are abstracted by type. The extra descriptions are extracted from parameter descriptions, as shown in the `DESCRIPTION` section in Figure 2.2, as well as from the `NAME` sections of each manual.

Additional details about the individual datasets are provided in Appendix A and Figure A.2. In terms of extracting the individual datasets, we build custom documentation parsers for each target project, which were usually applied over HTML renderings of each documentation set.

Py27 Dataset The Py27 dataset consists of 27 popular Python projects documented in English, as shown in Table 2.2. Datasets were extracted automatically using the `Zubr` toolkit detailed in Chapter 5.

Code Preprocessing Following standard practices in machine learning research on software (Allamanis et al., 2014), all component representations are linearized, lowercased and tokenized to remove constructs such as camelcase (e.g., `myFunction` to `my function`), underscores (e.g., `my_function` to `my function`), hyphens (e.g., `my-function` to `my function`). The idea is that by using subword tokens that results from this tokenization process, the model can then generalize across different function and variable names. To see more details about this conversion, all datasets and relevant code are publicly available at `https://github.com/yakazimir/Code-Datasets`.

2.4.2 Evaluation

For evaluation, we split all datasets into separate training, validation and test sets. For Java in the Stdlib collection, we reserve 60% of the data for training and the remaining 40% for validation (20%) and testing (20%). For all other datasets, we use a 70%-15%-15% split. From a retrieval perspective, these left out descriptions

Project	# Pairs	# Symbols	# Words	Vocab.
scapy	757	1,029	7,839	1,576
zipline	753	1,122	8,184	1,517
biopython	2,496	2,224	20,532	2,586
renpy	912	889	10,183	1,540
pyglet	1,400	1,354	12,218	2,181
kivy	820	861	7,621	1,456
pip	1,292	1,359	13,011	2,201
twisted	5,137	3,129	49,457	4,830
vispy	1,094	1,026	9,744	1,740
orange	1,392	1,125	11,596	1,761
tensorflow	5,724	4,321	45,006	4,672
pandas	1,969	1,517	17,816	2,371
sqlalchemy	1,737	1,374	15,606	2,039
pyspark	1,851	1,276	18,775	2,200
nupic	1,663	1,533	16,750	2,135
astropy	2,325	2,054	24,567	3,007
sympy	5,523	3,201	52,236	4,777
ipython	1,034	1,115	9,114	1,771
orator	817	499	6,511	670
obspy	1,577	1,861	14,847	2,169
rdkit	1,006	1,380	9,758	1,739
django	2,790	2,026	31,531	3,484
ansible	2,124	1,884	20,677	2,593
statsmodels	2,357	2,352	21,716	2,733
theano	1,223	1,364	12,018	2,152
nltk	2,383	2,324	25,823	3,151
sklearn	1,532	1,519	13,897	2,115

Table 2.2: A description of our Py27 dataset.

are meant to mimic unseen queries to our model. More generally, we can think of our evaluation as simulating a code retrieval system: given an input specification of a function, find the function signature that matches our specification (see Figure 2.4). After training our models, we evaluate on these held out sets by ranking all known components in each resource using Algorithm 2. A predicted component is counted as correct if it matches *exactly* a gold component.

We report the accuracy of predicting the correct representation at the first position in the ranked list (**Accuracy @1**) and within the top 10 positions (**Accuracy @10**). We also report the mean reciprocal rank **MRR**, or the multiplicative inverse of the rank of the correct answer as defined below (where $\text{rank}(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})$ denotes the rank of the correct component $\mathbf{z}$ given an input $\mathbf{x}$ at point d in the

Dataset	# Pairs	#Descr.	Symbols	Words	Vocab.
PHP_{fr}	6,155	14,058	7,922	70,800	5,904
PHP_{es}	5,823	13,285	7,571	69,882	5,790
PHP_{ja}	4,903	11,251	6,399	65,565	3,743
PHP_{ru}	2,549	6,030	3,340	23,105	4,599
PHP_{tr}	1,822	4,414	2,725	16,033	3,553
PHP_{de}	1,538	3,733	2,417	17,460	3,209

Table 2.3: The non-English PHP datasets.

dataset D):

$$\textbf{MRR} = \frac{1}{|D|}\sum_{d=1}^{|D|}\frac{1}{\text{rank}(\mathbf{x}^{(d)}, \mathbf{z}^{(d)})}$$

As an example, Figure 2.4 shows our model with the input *Gets the total cache size*, which corresponds to the PHP signature representation shown below:

```
int APCIterator::getTotalSize(void)
```

In our first SMT model, this signature is generated in the fourth position (or within the top 10 results, @10), whereas it is generated in the first position in the subsequent reranking model (related to @1).

Additional Baselines For comparison, we trained a bag-of-words classifier (the BOW Model in Table 2.4). This model uses the occurrence of word-component symbol pairs as binary features, and aims to see if word co-occurrence alone is sufficient to for ranking representations. In the setup described above, our discriminative models use more data than the baseline models in the form of additional textual descriptions and document-level features from elsewhere in each API, which therefore makes the results not directly comparable. We therefore train a more comparable translation model, shown as *M1 Descr.* in Table 2.4, by adding the additional textual data (i.e. parameter and return or module descriptions) to the model's parallel training data.

2.5 Experimental Results and Discussion

Main Results The full set of test results are shown in Table 2.4 across all datasets, including a summary of the results averaged over Stdlib and Py27 on the bottom. Across all 43 datasets and baseline models, IBM Model 1 (**IBM M1**) outperforms virtually all other models and is in general a strong baseline. Of particular note is the poor performance of the higher-order translation models based on Model 2 (**IBM M1**) and the Tree Model. While Model 2 is known to outperform Model 1 on more conventional translation tasks (Och and Ney, 2003), it appears that such improvements are not reflected in this type of semantic translation context.

Throughout all datasets, the bag-of-words (**BOW**) and Term Match (which is more of an information retrieval based method) baselines are outperformed by all other models. This shows that translation in this context is more complicated than simple word matching, and perhaps provides some justification more generally for treating the underlying task as a sequence prediction or SP problem. In some cases the term matching baseline is competitive with other models, suggesting that API collections differ in how language descriptions overlap with component names and naming conventions. It is clear, however, that this heuristic only works for English, as shown by results on the non-English PHP datasets and the generally lower number on the Stdlib dataset (i.e., average 12.8% Accuracy @1).

We achieve improvements on many datasets by adding additional data to the translation model (**M1 Descr.**). We achieve further improvements on all datasets using the discriminative model (**Reranker**), with most increases in performance occurring at how the top ten items are ranked. This last result suggests that phrase-level and document-level features can help to improve the overall ranking and translation, though in some cases the improvement is rather modest (see Appendix A for more analysis).

Discussion In addressing our initial research questions, we find that achieving high accuracy on these datasets is difficult. While the SMT models in particular perform rather well, there is a still much room for improvement, especially on achieving better Accuracy @1. While one might expect better results when mov-

Stdlib Test Results

Method	Java	Scheme	PHP_{en}	PHP_{fr}	Python	PHP_{es}
BOW Model	16.4 63.8 31.8	06.1 58.1 21.4	08.0 40.5 18.1	06.1 36.9 16.0	04.1 33.3 13.6	05.9 37.8 15.8
Term Match	15.7 41.3 24.8	25.5 61.2 37.4	15.6 37.0 23.1	04.0 15.8 07.7	16.6 41.7 24.8	02.9 10.4 05.4
IBM M1	34.3 79.8 50.2	32.1 75.5 46.2	35.5 70.5 47.2	32.1 65.1 43.5	22.7 61.0 35.8	29.5 63.7 41.2
IBM M2	30.3 77.2 46.5	29.5 71.4 43.9	33.2 67.7 45.0	30.6 62.2 41.2	21.4 58.0 34.4	26.7 59.8 38.3
Tree Model	29.3 75.4 45.3	26.1 71.2 40.3	28.0 63.2 39.8	27.9 59.3 38.6	17.5 55.4 30.7	25.9 61.0 37.6
M1 Descr.	33.3 77.0 48.7	33.1 75.5 47.1	34.1 71.1 47.2	31.0 64.8 42.7	22.7 62.3 35.9	28.6 64.9 41.1
Reranker	35.3 81.5 51.4	34.6 77.5 48.9	36.9 74.2 49.3	32.7 66.8 44.2	25.5 66.0 38.7	30.6 66.3 42.6

Method	Haskell	PHP_{ja}	Clojure	PHP_{ru}	Ruby	PHP_{tr}
BOW Model	05.6 55.6 21.7	04.7 33.2 13.8	03.0 49.2 16.4	04.4 43.6 16.6	07.0 38.0 16.9	05.4 43.4 17.6
Term Match	15.4 41.8 24.0	02.3 11.2 05.2	20.7 49.2 30.0	01.0 09.3 03.6	23.1 46.9 31.2	01.4 08.7 03.6
IBM M1	22.3 70.3 39.6	23.0 58.1 34.9	29.6 69.2 41.6	20.3 58.4 33.3	31.4 68.5 44.2	25.9 61.6 38.6
IBM M2	13.8 68.2 31.8	22.2 56.1 33.3	26.5 64.2 38.2	18.5 54.5 30.6	27.9 66.0 41.4	23.3 57.6 35.8
Tree Model	17.8 65.4 35.2	22.6 57.8 34.1	23.0 60.3 34.4	20.6 59.0 32.9	27.1 63.3 39.5	18,9 55.1 32.0
M1 Descr.	23.9 69.5 40.2	25.4 60.4 37.0	29.6 69.2 41.6	21.1 62.6 34.5	32.5 70.0 45.5	29.1 62.0 41.4
Reranker	24.7 73.9 43.0	25.8 61.8 37.8	35.0 76.9 47.9	21.1 66.8 35.9	35.1 72.5 48.0	29.9 63.8 41.2

Method	Elisp	PHP_{de}	C	Unix
BOW Model	09.9 54.6 23.5	04.3 39.2 15.3	08.8 48.8 20.0	08.6 49.6 21.0
Term Match	29.3 65.4 41.4	03.8 09.4 06.2	13.1 37.5 21.9	15.1 33.8 22.4
IBM M1	30.6 67.4 43.5	22.8 62.5 36.8	21.8 63.7 34.4	30.2 66.9 42.2
IBM M2	28.1 66.1 40.7	19.8 58.6 33.0	23.7 60.9 34.6	23.0 60.4 36.0
Tree Model	26.8 63.2 39.7	18.5 56.0 30.6	18.1 56.2 29.4	23.0 58.2 34.3
M1 Descr.	30.3 73.4 44.7	26.7 62.0 38.8	21.8 62.7 33.9	34.5 71.9 47.4
Reranker	37.6 80.5 53.3	28.0 65.9 40.5	29.7 67.4 40.1	34.5 74.8 48.5

Py27 Test Results

Method	renpy	zipline	biopython	kivy	ansible	pyglet
BOW Model	06.6 41.1 16.6	01.7 38.3 12.9	05.8 54.8 20.4	07.3 53.6 22.0	17.9 55.3 30.5	05.7 52.3 19.2
Term Match	25.7 59.5 38.7	28.5 50.8 36.2	23.5 48.1 31.7	30.0 62.6 41.3	24.8 54.0 35.8	20.4 50.9 31.2
IBM M1	25.0 62.5 37.4	23.2 58.0 36.2	29.6 75.6 46.2	35.7 70.7 47.4	36.7 72.3 48.8	26.6 70.9 41.5
IBM M2	23.5 52.9 34.4	23.2 50.8 34.1	27.2 75.4 43.7	28.4 65.0 41.7	31.1 65.7 42.8	23.3 62.8 35.7
M1 Descr.	30.8 61.7 42.0	27.6 62.5 40.7	29.6 75.6 45.8	33.3 67.4 45.3	35.5 71.6 47.5	26.1 69.5 41.3
Reranker	38.9 73.5 48.9	30.3 70.5 45.3	32.3 79.1 48.6	35.7 75.6 49.1	40.5 77.0 53.1	29.0 77.1 45.5

Method	rdkit	pip	twisted	vispy	orange	sympy
BOW	05.3 40.6 17.1	06.2 40.9 17.1	06.6 38.8 16.9	07.3 48.7 18.6	13.4 60.5 29.1	06.4 44.4 18.5
Term Match	13.3 46.6 23.9	19.1 50.2 30.7	17.6 44.1 26.2	29.2 64.0 41.1	37.9 69.7 49.3	20.2 44.9 28.8
IBM M1	22.6 58.0 35.2	15.0 56.4 29.4	26.8 61.8 38.8	29.8 68.2 43.4	40.3 79.8 53.7	32.6 70.5 45.2
IBM M2	16.0 48.0 27.5	15.0 48.7 26.5	22.4 55.9 34.0	20.7 63.4 34.6	31.2 71.1 45.1	28.6 65.4 40.9
M1 Descr.	25.3 60.6 37.2	18.6 56.4 32.3	27.7 61.4 39.4	28.6 70.1 42.3	40.3 78.3 54.0	32.8 70.2 45.5
Reranker	25.3 63.3 39.6	25.9 65.8 39.9	28.8 65.8 42.2	33.5 80.4 50.3	45.1 84.1 59.9	32.1 75.0 46.6

Method	pandas	sqlalchemy	pyspark	nupic	astropy	tensorflow
BOW	03.7 40.6 15.6	07.3 45.0 18.4	07.5 50.9 20.8	06.4 55.0 22.8	07.7 52.0 21.1	09.4 47.4 21.2
Term Match	19.3 43.7 27.9	17.3 48.4 26.6	20.5 46.9 29.1	23.6 51.0 33.1	26.1 49.1 34.3	25.2 48.7 33.5
IBM M1	28.4 63.3 40.5	24.2 71.1 39.7	34.6 78.7 50.1	30.1 71.4 44.0	27.8 66.9 41.2	34.3 70.7 47.4
IBM M2	25.0 58.3 36.8	20.0 62.3 34.3	33.9 74.3 48.1	22.4 67.0 38.4	25.0 62.6 37.7	30.7 66.3 43.2
M1 Descr.	29.1 62.7 41.0	28.8 70.3 43.0	37.1 78.7 52.1	30.9 69.8 44.6	30.7 66.6 43.4	35.3 71.5 48.0
Reranker	31.1 66.1 43.1	35.0 76.1 49.7	41.5 81.5 55.3	29.3 76.7 45.6	33.9 74.4 47.4	38.4 77.7 51.8

Method	ipython	orator	obspy	scapy	django	statsmodels
BOW	01.9 41.2 13.9	10.6 66.3 28.6	06.7 49.5 20.2	00.0 51.3 17.4	04.5 40.9 16.2	05.6 46.1 18.6
Term Match	23.8 56.7 33.8	31.9 64.7 43.7	19.9 46.6 30.0	21.2 43.3 28.7	19.3 48.0 29.1	16.7 39.9 25.1
IBM M1	23.8 61.9 36.3	31.1 78.6 46.2	32.6 72.4 47.1	17.6 61.0 33.5	23.6 57.8 35.5	25.2 64.0 37.9
IBM M2	20.6 52.9 32.8	23.7 70.4 38.3	29.6 64.8 41.7	16.8 58.4 31.0	20.5 53.3 31.0	19.8 61.4 33.6
M1 Descr.	24.5 59.3 36.5	32.7 79.5 47.5	33.8 75.8 48.3	20.3 61.9 34.7	22.9 57.8 34.6	25.4 64.8 37.8
Reranker	29.6 66.4 42.3	32.7 82.7 49.7	37.7 80.0 52.3	21.2 67.2 37.2	25.8 64.5 39.4	28.8 69.1 41.7

Method	theano	nltk	sklearn
BOW	03.2 43.7 16.2	05.0 44.2 16.3	05.2 45.8 17.7
Term Match	16.3 37.1 24.0	19.8 45.6 28.4	24.4 50.6 32.5
IBM M1	22.9 56.2 35.2	21.8 64.7 35.6	26.2 65.9 39.0
IBM M2	22.4 50.8 33.8	18.4 59.1 31.3	20.9 61.5 34.1
M1 Descr.	26.2 58.4 37.8	28.2 68.0 41.5	27.9 67.6 41.3
Reranker	27.3 66.1 39.9	31.6 72.5 45.7	29.2 75.5 44.5

Results Summary (averaged over all datasets)

Method	Stdlib	Py27
BOW	6.8 45.4 18.7	6.5 47.8 19.5
Term Match	12.8 32.5 19.5	22.9 50.6 32.4
IBM M1	27.8 66.4 40.8	27.8 67.1 41.2
IBM M2	24.9 63.1 37.8	23.8 61.1 36.6
M1 Descr.	28.6 67.5 41.7	29.3 67.4 42.5
Reranker	31.1 71.0 44.5	32.3 73.5 46.5

Accuracy @1	Accuracy @10	Mean Reciprocal Rank (MRR)

Table 2.4: Test results for the Stdlib and Py27 datasets according to Accuracy@1, Accuracy@10 and MRR.

ing from a word-based model to a model that exploits phrase and hierarchical phrase features (as used in the reranker model), the sparsity of the component vocabulary is such that most phrase patterns in the training are not observed in the evaluation. In many benchmark SP datasets, such sparsity issues do not occur (Cimiano and Minock, 2010), suggesting that state-of-the-art methods will have similar problems when applied to our datasets. Regarding the question of whether adding background knowledge about the API helps to build more robust translation models, we see that this can help to increase accuracy via the reranker model, though the gains are rather modest.

We note that pure SMT approaches, including the SMT methods pursued here, are somewhat non-standard relative to other methods used in SP (see review in Chapter 1). We believe that the main reason for this is that the decoding problem is considerably harder to solve for SP as compared with ordinary machine translation, since SP requires translating to a correct and well-defined formal language. In work that uses SMT for SP (Andreas et al., 2013; Haas and Riezler, 2016), post-processing tricks are often employed at decoding time (e.g., pruning out candidate translations in a k best list that are ungrammatical) that failed to work in our setting. Under our approach, the underlying decoding strategy is modified to consider only valid output translations, in order to solve a problem that we define in the following way:

- **Constrained Decoding:** How can we ensure that our MT models guarantee well-formed SP output at test/decoding time?

We found alternative grammar-based SP approaches (e.g., Zettlemoyer and Collins (2009); Börschinger et al. (2011)) to be equally problematic in our setting, given that such methods rely on rule extraction techniques (i.e., to construct the underlying semantic grammars) that are hard to scale to domains larger than those encountered in benchmark tasks (cf. Cai and Yates (2013)). Such methods also often assume a one-to-one mapping between words and semantic concepts (often taking the form of pre-terminal rules), which is hard to deal with in the code setting where certain constructs (e.g., variables, namespace information) remain implicit in the text descriptions (cf. Quirk et al. (2015)).

In general, we see two possible use cases for this data. First, for benchmarking SP

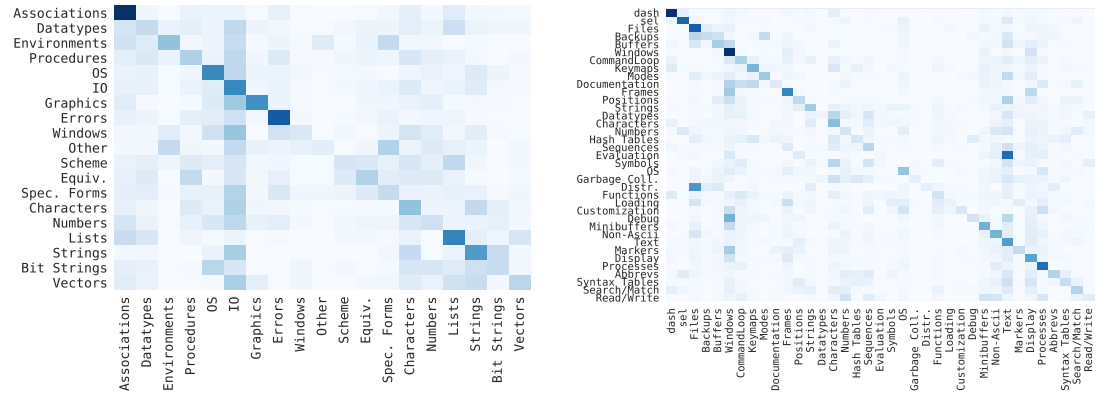


Figure 2.8: Erroneous function predictions by documentation category for Scheme and Elisp.

models on the task of semantic translation. While there has been a trend towards learning executable semantic parsers (Berant et al., 2013; Liang, 2016), there has also been renewed interest in supervised learning of formal representations in the context of neural SP models (Dong and Lapata, 2016; Jia and Liang, 2016). We believe that good performance on our datasets should lead to better performance on more conventional SP tasks, and raise new challenges involving sparsity and multilingual learning.

Model Errors We performed analysis on some of the incorrect predictions made by our models. For some documentation sets, such as those in the GNU documentation collection, information is organized into a small and concrete set of categories/chapters, each corresponding to various features or modules in the language and related functions. Given this information, Figure 2.8 shows the confusion between predicting different categories of functions, where the rows show the categories of functions to be predicted and the columns show the different categories predicted. We built these plots by finding the categories of the top 50 non-gold (or erroneous) representations generated for each validation example.

The step-like lines through the diagonal of both plots show that alternative predictions (shaded according to occurrence) are often of the same category, most

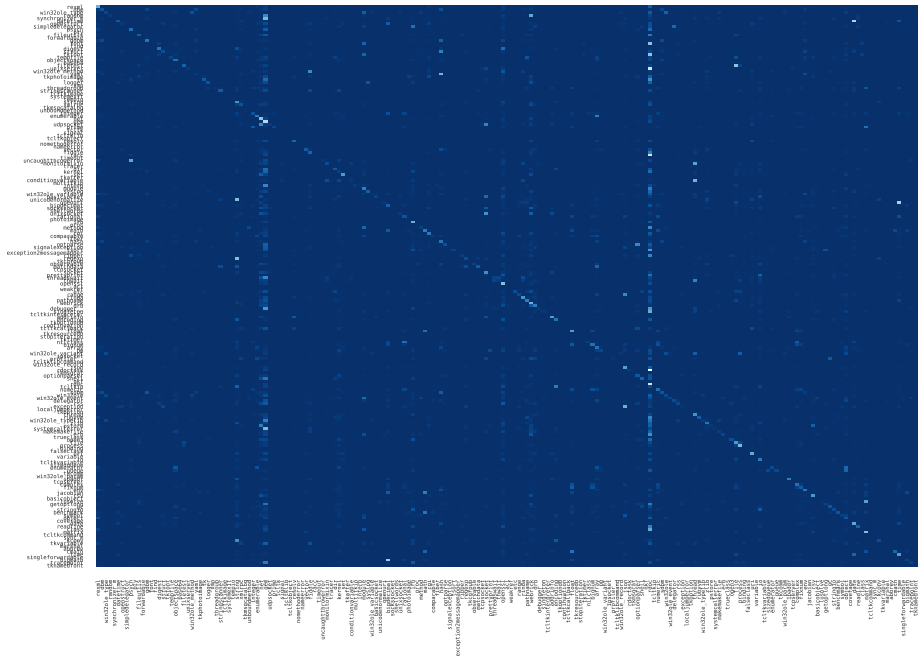


Figure 2.9: General erroneous function prediction trend for the Ruby standard library (with lighter colors indicating stronger confusion).

strikingly for the corner categories. This trend seems stable across other datasets, even among datasets with large numbers of categories (see Figure 2.9 for the same trend observed in the Ruby standard library). Interestingly, many confusions appear to be between related categories. For example, when making predictions about `Strings` functions in Scheme, the model often generates functions related to `BitStrings`, `Characters` and `IO`. Again, this trend seems to hold for other documentation sets, suggesting that the models are often making semantically sensible decisions.

Looking at errors in other datasets, one common error involves generating functions with the same name and/or functionality. In large libraries, different modules sometimes implement the same core functions, such the `genericpath` or `posixpath` modules in Python. When generating a representation for the text return size of file, our model confuses the `getsize(filename)` function in one module with others. Similarly, other subtle distinctions that are not explicitly ex-

pressed in the text descriptions are not captured, such as the distinction in Haskell between *safe* and *unsafe* bit shifting functions.

2.6 Conclusions

In this chapter, we introduced 43 new datasets for SP development based on the standard library documentation for 10 popular programming languages and number of open source Python projects. As part of an effort to address the resource problem for SP, we propose using such source code documentation as a kind of parallel translation corpus, consisting of text description and code pairs (i.e., formal representations of functions, or *function signatures*), the latter of which we regard a proxies for logical forms. In order to establish baseline results of our new text to function signature SP task, we introduce several new models based largely on SMT. These baselines indicate that the task is not easy, and while simple SMT models with specialized decoders tend to perform best, there is still a lot of room for improving accuracy. Relative to other benchmark tasks, these datasets appear to raise new challenges largely related to the large scope and sparsity of the target code vocabulary, which appear to create challenges for more sophisticated SMT models and SMT-based decoding to SP more generally.

In contrast to traditional SP tasks, it must be emphasized that the target code representations being learned deviate in several important respects from traditional logical forms. Chief among these differences is that the target representations have limited compositionality and lack a clearly defined semantics. This is not altogether problematic, since our main idea is to use these representations (which resemble atomic predicate logic representations) to study the more general translation and lexicon learning problem encountered in SP. Nonetheless, given that these signature representations are formal languages, we can further formalize them and define a formal semantics in terms of classical logic, which is an idea that we pursue in Richardson (2018) and discuss in Chapter 5.

One exciting aspect of the source code domain is that much of the declarative knowledge needed for doing deep reasoning about code can be easily extracted directly from information in the underlying libraries (e.g., information about types, class hierarchies, related functions). For this reason, we see source code libraries

as an interesting domain for experimenting with end-to-end SP and reasoning (i.e., according to the NLU model introduced in Chapter 1). Under this approach, we might further think of a source code library as a kind of knowledge base for reasoning. Our working assumption, however, is that any such efforts at deep NLU first relies on being able to robustly translate text to the underlying source code representations, which is the main task investigated in this chapter. Given that the proposed translation models still have a lot of room for improvement, we return to this underlying task in the next chapter and explore the idea of training SPs on multiple code datasets as way to improve on the baselines introduced here.

3 Polyglot Semantic Parsing

3.1 Learning from Multiple Datasets

3.1.1 The Idea

In the previous chapter, we considered the problem of translating source code documentation to lower-level code template representations as part of an effort to model the meaning of such documentation. Example documentation for a number of programming languages is shown in Figure 3.1, where each *docstring* description in red describes a given function (blue) in the library. For example, given the description *Returns the greater of two long values* (as already introduced), we want to translate this to the following code representation:

```
public static long max(long a, long b)
```

While capturing the semantics of docstrings is in general a difficult task, learning the translation from descriptions to formal code representations (e.g., formal representations of functions) is proposed as a reasonable first step towards learning more general natural language understanding models in the software domain. Under this approach, one can view a software library, or API, as a kind of parallel translation corpus for studying *text* → `code` or `code` → *text* translation.

In pursuing this idea, we extracted the standard library documentation for 10 popular programming languages across a number of natural languages to study the problem of text to function signature translation. Initially, these datasets were proposed as a resource for studying semantic parser induction (Mooney, 2007b), or for building models that learn to translate text to formal meaning representations from parallel data. In our follow-up work (Richardson and Kuhn, 2017a), we also look at using the resulting models to do automated question-answering (QA) and code retrieval on target APIs (a topic that we discuss more in Chapter 5), and experimented with an additional set of software datasets built from 27 open-source Python projects.

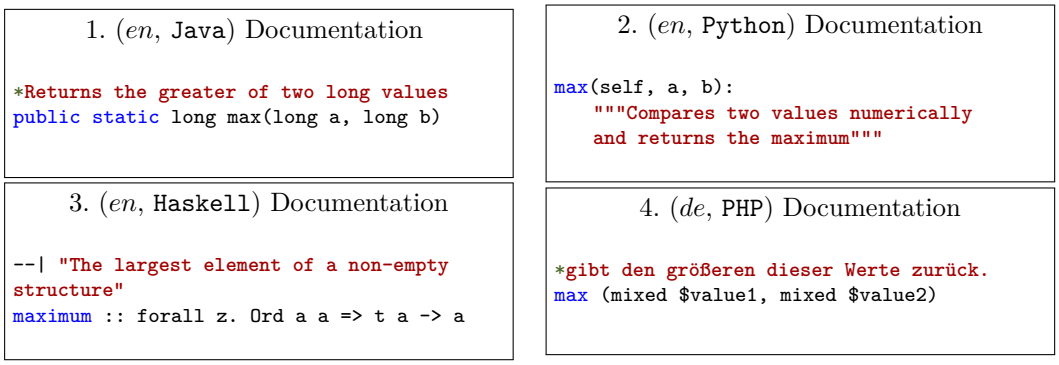


Figure 3.1: Example source code documentation.

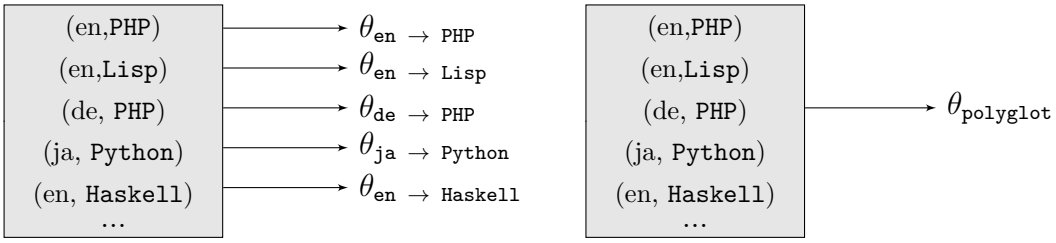


Figure 3.2: Building individual API models (left) versus polyglot modeling (right).

As traditionally done in semantic parsing (Zettlemoyer and Collins, 2012), the approach we introduced so far involves learning individual models for each parallel dataset or language pair, e.g., (*en*, `Java`), (*de*, `PHP`), and (*en*, `Haskell`), as illustrated on the left side of Figure 3.2. Looking again at the examples in Figure 3.1, we notice that while programming languages differ in terms of representation conventions, there is often overlap between the functionality implemented and naming in these different languages (e.g., the `max` function), and redundancy in the associated linguistic descriptions. In addition, each English description (Figure 3.1.1-4) describes `max` differently using the following synonyms:

greater, maximum, largest

In this case, it would seem that training models on multiple datasets, as opposed to single language pairs, might make learning more robust, and help to capture var-

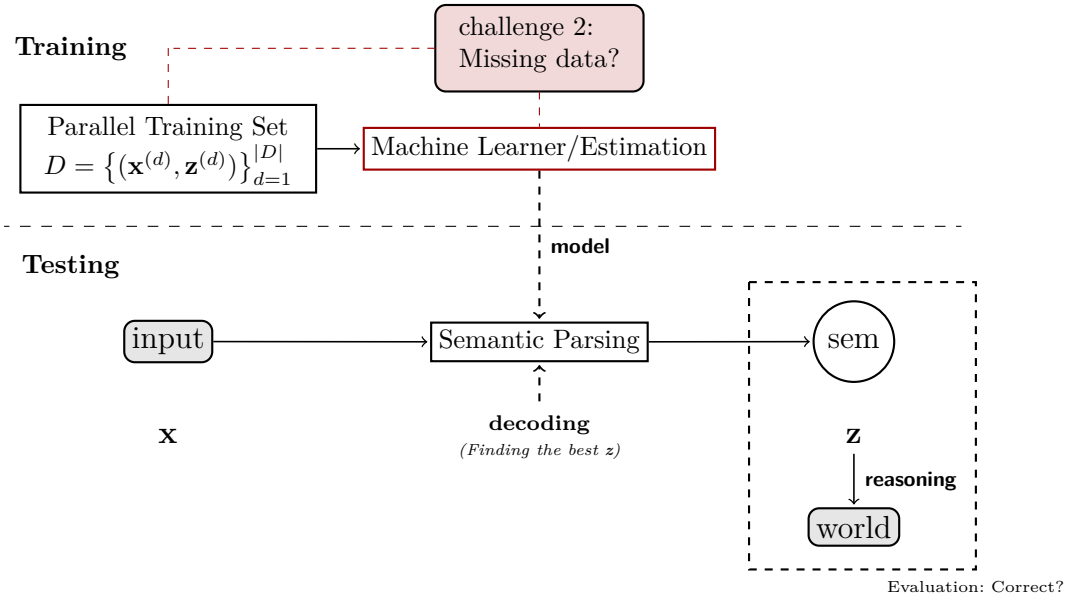


Figure 3.3: The standard semantic parsing setup and the second resource challenge (from Section 1.3).

ious linguistic alternatives. In this chapter, we investigate the following question: does training semantic parsers on multiple datasets indeed lead to more robust translation models?

3.1.2 Addressing a Different Resource Problem

With the software QA application in mind, one limitation of our initial approach is that it doesn't allow one to freely translate a given description to multiple output programming languages, which would be useful for comparing how different programming languages represent the same functionality. The model also cannot translate between natural languages and programming languages that are not observed during training (what we refer to in Chapter 1 as *missing data*). While software documentation is easy to find in bulk, if a particular API is not already documented in a language other than English (e.g., `Haskell` in German or *de*), it is unlikely that such a translation will appear without considerable effort by

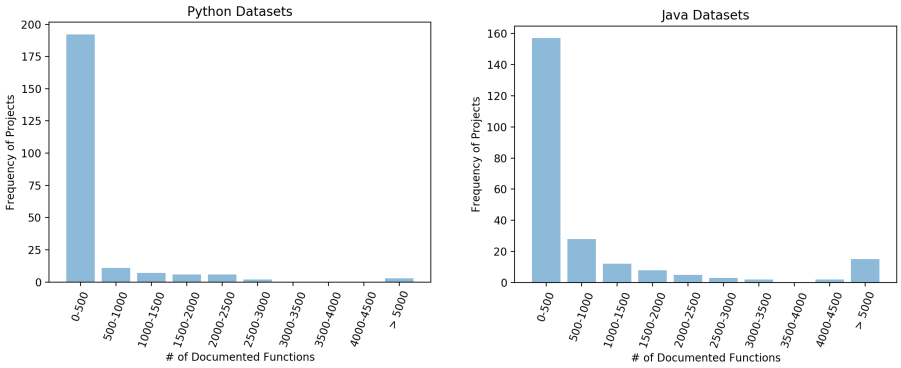


Figure 3.4: The amount of documentation extracted from 461 Python and Java projects from GitHub.com.

experienced translators.

Similarly, many individual APIs may be too small or poorly documented to build individual models or QA applications, and will in some way need to bootstrap off of more general models or resources. More generally, finding targeted data specific to a particular source code project or programming language can sometimes be difficult. To quantify this, we show in Figure 3.4 the size of different datasets constructed from 461 influential Python and Java source code projects hosted on GitHub. While we would ideally want to find datasets that contains thousands of documented functions, we find that most projects contain 500 or less data points.

To deal with these issues, which are again summarized in Figure 3.3 using the setup from Chapter 1, we aim to learn more general text-to-code translation models that are trained on multiple datasets simultaneously, as shown in Figure 3.2. Our ultimate goal is to build *polyglot* translation models (cf. Johnson et al. (2016)), or models with shared representations that can translate any input text to any output programming language, regardless of whether such language pairs were encountered explicitly during training. We specifically ask the following research questions:

- **Multiple Datasets**: Does training semantic parsing models on multiple datasets lead to more robust translation models?

- **Language crossing**: Can we learn to translate between different language pairs, including pairs not observed during training?

Inherent in this task is the challenge of building an efficient polyglot decoder, or a translation mechanism that allows such crossing between input and output languages. A key challenge is ensuring that such a decoder generates well-formed code representations, which is not guaranteed when one simply applies standard decoding strategies from SMT and neural MT (cf. Cheng et al. (2017); Krishnamurthy et al. (2017)). Given our ultimate interest in API QA, such a decoder must also facilitate monolingual translation, or being able to translate to specific output languages as needed. This issue, which we first discussed in the previous chapter, concerns the following general problem:

- **Constrained Decoding**: How can we control the output space for SP decoding and ensure well-formedness of the output?

To solve this constrained decoding problem, we introduce a new graph-based decoding and representation framework that reduces to solving shortest path problems in directed graphs. We investigate several translation models that work within this framework, including traditional SMT models and models based on neural networks, and report state-of-the-art results on datasets introduced in the last chapter. To show the applicability of our approach to more conventional SP tasks, we apply our methods to the GeoQuery domain (Zelle and Mooney, 1996) and the Sportscaster corpus (Chen et al., 2010). These experiments also provide insight into the main technical documentation task and highlight the strengths and weaknesses of the various translation models being investigated.

3.2 Related Work

Our approach builds on the baseline models introduced in the previous chapter. This work is positioned within the broader semantic parsing (SP) literature, where traditionally SMT (Wong and Mooney, 2006) and parsing (Zettlemoyer and Collins, 2009) methods are used to study the problem of translating text to formal meaning representations, usually centering around QA applications (Berant et al.,

2013). More recently, there has been interest in using neural network approaches either in place of (Dong and Lapata, 2016; Kočiský et al., 2016) or in combination with (Misra and Artzi, 2016; Jia and Liang, 2016; Cheng et al., 2017) these traditional models, the latter idea we look at in this chapter.

Work in NLP on software documentation has accelerated in recent years due in large part to the availability of new data resources through websites such as StackOverflow and GitHub (cf. Allamanis et al. (2017)). Most of this recent work focuses on processing large amounts of API data in bulk (Gu et al., 2016; Miceli Barone and Sennrich, 2017), either for learning longer executable programs from text (Yin and Neubig, 2017; Rabinovich et al., 2017), or solving the inverse problem of code to text generation (Iyer et al., 2016; Richardson et al., 2017). In contrast to our work, these studies do not look explicitly at translating to target APIs, or at non-English documentation.

The idea of polyglot modeling has gained some traction in recent years for a variety of problems (Tsvetkov et al., 2016) and has appeared within work in SP under the heading of *multilingual SP* (Jie and Lu, 2014; Duong et al., 2017). A related topic is learning from multiple knowledge sources or domains (Herzig and Berant, 2017), which is related to our idea of learning from multiple APIs. When building models that can translate between unobserved language pairs, we use the term *zero-shot translation* from Johnson et al. (2016).

3.3 Baseline Semantic Translator

Problem Formulation Throughout the chapter, we refer to target code representations as API *components*. In all cases, components will consist of formal representations of functions, or function signatures, e.g.

```
long max(int a, int b)
```

which include a function name (`max`), a sequence of arguments (`int a, int b`), and other information such as a return value (`long`) and namespace (for more details, see Richardson (2018)). For a given API training dataset $D = \{(\mathbf{x}_i, \mathbf{z}_i)\}_{i=1}^{d}$ of size d, the goal is to learn a model that can generate *exactly* a correct com-

ponent sequence $\mathbf{z} = (z_1, .., z_{|\mathbf{z}|})$, within a finite space $\mathcal{C}$ of signatures (i.e., the space of all defined functions), for each input text sequence $\mathbf{x} = (x_1, ..., x_{|\mathbf{x}|})$. This involves learning a probability distribution $p(\mathbf{z} \mid \mathbf{x})$. As such, one can think of this underlying problem as a *constrained* MT task.

In this section, we describe the baseline approach from the last chapter. Technically, our initial approach has two components: a simple word-based translation model and task specific decoder, which is used to generate a k-best list of candidate component representations for a given input $\mathbf{x}$. We then use a discriminative model to rerank the translation output using additional non-world level features. The goal in this section is to provide the technical details of our translation approach, which we improve in Section 3.4.

3.3.1 Word-based Translation Model

The translation models investigated in the last chapter use a noisy-channel formulation where $p(\mathbf{z} \mid \mathbf{x}) \propto p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})$ via Bayes rule. By assuming a uniform prior on output components, $p(\mathbf{z})$, the model therefore involves estimating $p(\mathbf{x} \mid \mathbf{z})$, which under a word-translation model is computed using the following formula:

$$p(\mathbf{x} \mid \mathbf{z}) = \sum_{a \in \mathcal{A}} p(\mathbf{x}, a \mid \mathbf{z})$$

where the summation ranges over the set of all many-to-one word alignments $\mathcal{A}$ from $\mathbf{x} \rightarrow \mathbf{z}$, with $|\mathcal{A}|$ equal to $(|\mathbf{z}|+1)^{|\mathbf{x}|}$. We investigated various types of sequence-based alignment models (Och and Ney, 2003), and find that the classic IBM Model 1 outperforms more complex word models. This model factors in the following way and assumes an *independent word generation* process:

$$p(\mathbf{x} \mid \mathbf{z}) = \frac{1}{|\mathcal{A}|} \prod_{j=1}^{|\mathbf{x}|} \sum_{i=0}^{|\mathbf{z}|} p_t(x_j \mid z_i) \tag{3.1}$$

where each p_t defines a multinomial distribution over a given component term z for all words x.

The decoding problem for the above translation model involves finding the most likely output $\hat{z}$, which requires solving an $\arg\max_{\mathbf{z}}$ over Equation 3.1. In the general

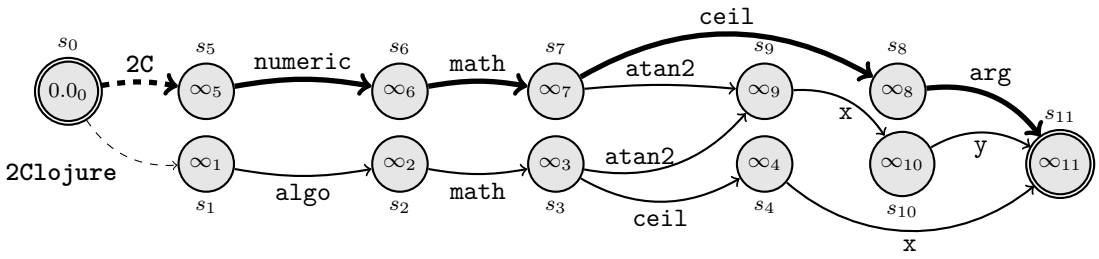


Figure 3.5: A DAFSA representation for a portion of the component sequence search space $\mathcal{C}$ that includes math functions in **C** and **Clojure**, and an example path/translation (in **bold**): **2C** `numeric math ceil arg`.

case, this problem is known to be $\mathcal{NP}$-complete for the models under consideration (Knight, 1999) largely due to the large space of possible predictions $\mathbf{z}$. We avoided these issues by exploiting the finiteness of the target component search space (an idea we also pursue here and discuss more below), and developed a constrained decoding algorithm that runs in time $O(|\mathcal{C}| \log |\mathcal{C}|)$ (i.e., the RankDecoder in Algorithm 2). While this works well for small APIs, it becomes less feasible when dealing with large sets of APIs, as in the polyglot case, or with more complex semantic languages typically used in SP (see Liang (2013)).

3.4 Shortest Path Framework

To improve the baseline translation approach used previously, we pursue a graph based approach. Given the formulation above and the finiteness of our prediction space $\mathcal{C}$, our approach exploits the fact that we can represent the complete component search space for any set of APIs as a directed acyclic finite-state automaton (DAFSA), such as the one shown graphically in Figure 3.5. The underlying graph is constructed by concatenating all of the component representations for each API of interest and applying standard finite-state construction and minimization techniques (Mohri, 1996). Each path in the resulting compact automaton is therefore a well-formed component representation.

Using an idea from Johnson et al. (2016), we add to each component representa-

Algorithm 4 Standard DAG Shortest-Path Search Algorithm

Input: Labeled DAG $\mathcal{G} = (V, E)$, weight function w, source node b
Output: Shortest path

```
1: d[V[G]] ← ∞                                  ▷ Initializes Shortest-Path estimate at each node
2: π[V[G]] ← Nil                                ▷ Initializes backwards pointers
3: d[b] ← 0                                     ▷ Initializes source node score to 0
4: for each node u ≥ b ∈ V[G] in sorted order do   ▷ Search graph adjacency
5:     for each labeled edge (v, z) ∈ Adj[u] do
6:         if d[v] > d[u] + w(u, v, z) then     ▷ The relaxation step
7:             d[v] ← d[u] + w(u, v, z)         ▷ Update score to this shortest path
8:             π[v] ← u                         ▷ Record the node with current shortest path
9: return FindPath(π, |V|, b)                   ▷ Retrieve the shortest path by backtracking
```

tion an *artificial* token that identifies the output programming language or library. For example, the two edges from the initial state 0 in Figure 3.5 are labeled as *2C* and *2Clojure*, which identify the C and Clojure programming languages respectively. All paths starting from the right of these edges are therefore valid paths in each respective programming language. The paths starting from the initial state 0, in contrast, correspond to all valid component representations in all languages.

Decoding reduces to the problem of finding a path for a given text input **x**. For example, given the input *the ceiling of a number*, we would want to find the paths corresponding to the following component translations:

2C	`numeric math ceil arg`
2Clojure	`algo math ceil x`

in the graph shown in Figure 3.5. Using the trick above, our setup facilitates both monolingual decoding, i.e., generating components specific to a particular output language (e.g., the C language via the path shown in bold), and polyglot decoding, i.e., generating any output language by starting at the initial state 0 (e.g., to C and Clojure languages).

We formulate the decoding problem using a variant of the well-known single source shortest path (SSSP) algorithm for directed acyclic graphs (`DAG`s) (Johnson (1977)). This involves a graph $\mathcal{G} = (V, E)$ (nodes V and labeled edges E, see graph in Figure 3.5), and taking an off-line topological sort of the graph's vertices.

Using a data structure $d \in \mathbb{R}^{|V|}$ (initialized as $\infty^{|V|}$, as shown in Figure 3.5), the standard SSSP algorithm (which is the *forward update* variant of the Viterbi algorithm (Huang, 2008); see Algorithm 4) works by searching forward through the graph in sorted order (starting line 4) and finding for each node v an incoming labeled edge u, with label z, that solves the following recurrence (line 6):

$$d(v) = \min_{(u,z):(u,v,z)\in E} \Big\{ d(u) + w(u, v, z) \Big\} \tag{3.2}$$

where $d(u)$ is shortest path score from a unique source node b to the incoming node u (computed recursively) and $w(u, v, z)$ is the weight of the particular labeled edge. The weight of the resulting shortest path is commonly taken to be the sum of the path edge weights as given by w, and the output translation is the sequence of labels associated with each edge. This algorithm runs in linear time over the size of the graph's adjacency matrix (`Adj`) and can be extended to find k SSSPs. In the standard case, a weighting function w is provided by assuming a static weighted graph. In our translation context, we replace w with a translation model, which is used to dynamically generate edge weights during the SSSP search for each input $\mathbf{x}$ by scoring the translation between $\mathbf{x}$ and each edge label z encountered.

Given this general framework, many different translation models can be used for scoring. In what follows, we describe two types of decoders based on lexical translation (or unigram) models and neural sequence models. Technically, each decoding algorithm involves modifying the standard SSSP search procedure by adding an additional data structure s to each node (see Figure 3.5), which is used to store information about translations (e.g., running lexical translation scores, RNN state information) associated with particular shortest paths. By using these two very different models, we can get insight into the challenges associated with the technical documentation translation task. As we show in Section 3.6, each model achieves varying levels of success when subjected to a wider range of SP tasks, which reveals differences between our task and other SP tasks.

Algorithm 5 Lexical Shortest Path Search

Input: Input $\mathbf{x}$ of size n, DAG $\mathcal{G} = (V, E)$, lexical translation function p_t, source node b with initial score o.
Output: Shortest component path

1: $d[V[\mathcal{G}]] \leftarrow \infty, \pi[V[\mathcal{G}]] \leftarrow Nil, d[b] \leftarrow o$
2: $s[V[\mathcal{G}], n] \leftarrow 0.0$ ▷ Shortest path sums at each node
3: **for** each vertex $u \geq b \in V[\mathcal{G}]$ in sorted order **do**
4: **for** each vertex and label $(v, z) \in \texttt{Adj}[u]$ **do**
5: $\texttt{score} \leftarrow -log\big[\ \prod_i^n p_t(x_i \mid z) + s[u, i]\ \big]$
6: **if** $d[v] > \texttt{score}$ **then**
7: $d[v] \leftarrow \texttt{score}, \pi[v] \leftarrow u$
8: **for** i in $1, .., n$ **do** ▷ Update scores
9: $s[v, i] \leftarrow p_t(x_i \mid z) + s[u, i]$
10: **return** FindPath$(\pi, |V|, b)$

3.4.1 Lexical Translation Shortest Path

In our first model, we use the lexical translation model and probability function p_t in Equation 3.1 as the weighting function, which can be learned efficiently offline using the EM algorithm (see details in Section 2.3.1). When attempting to use the SSSP procedure to compute this equation for a given source input $\mathbf{x}$, we immediately have the problem that such a computation requires a complete component representation. As Knight and Al-Onaizan (1998) observe:

> Notice that in .. [Equation 3.1].. there is no notion of consuming the source sentence word by word and producing the target sentence. Instead, all *source words must remain available for consultation*.... These properties make [IBM] Model 1 unattractive for finite-state [and graph] modeling.

We use an approximation that involves ignoring the normalizer $|\mathcal{A}|$ and exploiting the word independence assumption of the model, which allows us to incrementally compute translation scores for individual source words given output translations corresponding to shortest paths during the SSSP search.

The full decoding algorithm in shown in Algorithm 5, where the red highlights the adjustments made to the standard SSSP search as presented in Cormen et al. (2009) (shown in Algorithm 4). The main modification involves adding a data

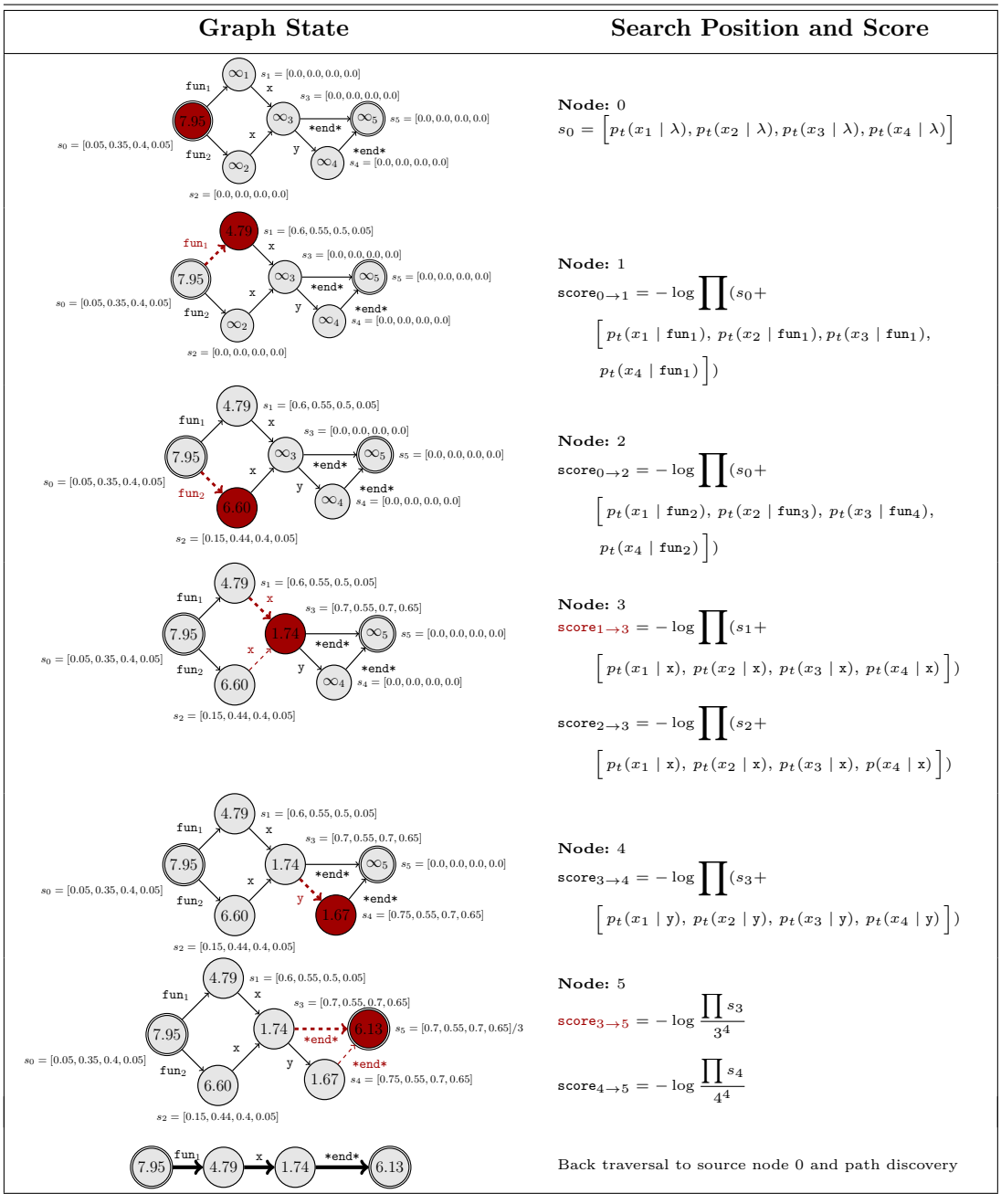
structure $s \in \mathbb{R}^{|V| \times |\mathbf{x}|}$ (initialized as $0.0^{|V| \times |\mathbf{x}|}$ at line 2) that stores a running sum of source word scores given the best translations at each node, which can be used for computing the inner sum in Equation 3.1. For example, given an input utterance *ceiling function*, s_6 in Figure 3.5 contains the *independent* translation scores for words *ceiling* and *function* given the edge label `numeric` and p_t. Later on in the search, these scores are used to compute s_7, which will provide translation scores for each word given the edge sequence *numeric math*. Taking the product over any given s_j (as done in line 7 to get `score`) will give the probability of the shortest path translation at the particular point j. Here, the transformation into $-\log$ space is used to find the *minimum* incoming path. Standardly, the data structure π (or *predecessor*) can be used to retrieve the shortest path back to the source node b (done via the FindPath method). An illustration of an example run of this algorithm is shown in Figure 3.6.

3.4.2 Neural Shortest Path

Our second set of models use neural networks to compute the weighting function in Equation 3.2 (for a general overview of neural networks and the notation used below, see Appendix B). We use an encoder-decoder model with global attention (Bahdanau et al., 2014; Luong et al., 2015), which has the following two components (as shown in Figure 3.7):

Encoder Model The first is an *encoder* network, which uses a bi-directional recurrent neural network (RNN) architecture with LSTM units (Hochreiter and Schmidhuber, 1997) to compute a sequence of forward annotations or hidden states $(\overrightarrow{h}_1, ..., \overrightarrow{h}_{|\mathbf{x}|})$ and a sequence of backward hidden states $(\overleftarrow{h}, ..., \overleftarrow{h}_{|\mathbf{x}|})$ for the input sequence $(x_1, ..., x_{|\mathbf{x}|})$. Each word is then represented as the concatenation of its forward and backward states:

$$h_j = \left[\overrightarrow{\texttt{LSTM}}(\overrightarrow{h}_{j-1}, \mathbf{E}^{in}_{x_j}) \; ; \; \overleftarrow{\texttt{LSTM}}(\overleftarrow{h}_{j+1}, \mathbf{E}^{in}_{x_j})\right]$$



	$p_t(\cdot \mid \text{fun}_1)$	$p_t(\cdot \mid \text{fun}_2)$	$p_t(\cdot \mid \text{x})$	$p_t(\cdot \mid \text{y})$	$p_t(\cdot \mid \lambda)$
function_1	0.6	0.1	0.1	0.05	0.05
function_2	0.1	0.8	0.1	0.05	0.05
applied	0.2	0.1	0.0	0.0	0.35
to	0.1	0.0	0.2	0.0	0.4
arg_x	0.0	0.0	0.6	0.0	0.05
and	0.0	0.0	0.0	0.2	0.05
arg_y	0.0	0.0	0.0	0.7	0.05

Figure 3.6: An illustration of the lexical SSSP algorithm for the text input $\mathbf{x}$ = *function_1 applied to arg_x*. The table for p_t is on the bottom, where λ denotes an artificial NULL word token on the target side.

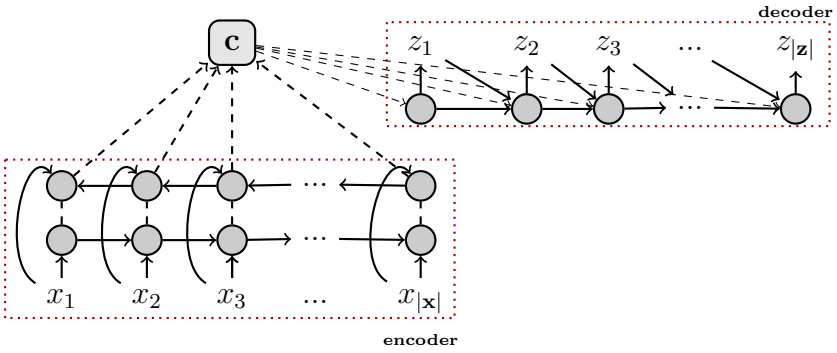


Figure 3.7: An illustration of the encoder decoder model architecture.

Decoder Model The second component is a *decoder* network, which directly computes the conditional distribution $p(\mathbf{z} \mid \mathbf{x})$ as follows:

$$p(\mathbf{z} \mid \mathbf{x}) = \sum_{i=1}^{|\mathbf{z}|} \log p_\Theta(z_i \mid z_{<i}, \mathbf{x}) \tag{3.3}$$

$$p_\Theta(z_i \mid z_{<i}, \mathbf{x}) \sim \texttt{softmax}(f(\Theta, z_{<i}, \mathbf{x})) \tag{3.4}$$

where f is a non-linear function that encodes information about the sequence $z_{<i}$ (i.e., all decisions $z_0, ..., z_{i-1}$ made previously) and the input $\mathbf{x}$ given the model parameters Θ. We can think of this model as an ordinary RNN language model that is additionally conditioned on the input $\mathbf{x}$ using information from our encoder. We implement the function f in the following way:

$$f(\Theta, z_{<i}, \mathbf{x}) = \mathbf{W}_o \eta_i + \mathbf{b}_o \tag{3.5}$$

$$\eta_i = \texttt{MLP}(c_i, g_i) \tag{3.6}$$

$$g_i = \texttt{LSTM}_{dec}(g_{i-1}, \mathbf{E}^{out}_{z_{i-1}}, c_i) \tag{3.7}$$

where MLP is a multi-layer perceptron model with a single hidden layer, $\mathbf{E}^{out} \in \mathbb{R}^{|\Sigma_{dec}| \times e}$ is a randomly initialized embedding matrix, g_i is the decoder's hidden state at step i, and c_i is a context-vector that encodes information about the input $\mathbf{x}$ and the encoder annotations. Each context vector c_i in turn is a weighted sum

of each annotation h_j against an attention vector $\alpha_{i,j}$, or $c_i = \sum_{j=1}^{|\mathbf{x}|} \alpha_{i,j} h_j$, which is jointly learned using an additional single layered multi-layer perceptron defined in the following way:

$$\alpha_{i,j} \propto \exp(e_{i,j}) \tag{3.8}$$

$$e_{i,j} = \texttt{MLP}(g_{i-1}, h_j) \tag{3.9}$$

Lexical Bias and Copying In contrast to standard MT tasks, we are dealing with a relatively low-resource setting where the sparseness of the target vocabulary is an issue. For this reason, we experimented with integrating lexical translation scores using a biasing technique from Arthur et al. (2016). Their method is based on the following computation for each token z_i:

$$\mathbf{bias}_i = \begin{bmatrix} p_{t'}(z_1 \mid x_1) & \dots & p_{t'}(z_1 \mid x_{|\mathbf{x}|}) \\ \vdots & \ddots & \vdots \\ p_{t'}(z_{|\Sigma_{dec}|} \mid x_1) & \dots & p_{t'}(z_{|\Sigma_{dec}|} \mid x_{|\mathbf{x}|}) \end{bmatrix} \begin{bmatrix} \alpha_{i,1} \\ \vdots \\ \alpha_{i,|\mathbf{x}|} \end{bmatrix}$$

The first matrix uses the inverse ($p_{t'}$) of the lexical translation function p_t already introduced to compute the probability of each word in the target vocabulary Σ_{dec} (the columns) with each word in the input $\mathbf{x}$ (the rows), which is then weighted by the attention vector from Equation 3.8. $\mathbf{bias}_i$ is then used to modify Equation 3.5 in the following way:

$$f_{bias}(\Theta, z_{<i}, \mathbf{x}) = \Big[\mathbf{W}_o \eta_i + \mathbf{b}_o\Big] + \log(\texttt{bias}_i + \epsilon)$$

where ϵ is a hyper-parameter that helps to preserve numerical stability and biases more heavily on the lexical model when set lower.

We also experiment with the *copying* mechanism from Jia and Liang (2016), which works by allowing the decoder to choose from a set of actions, a_j, that includes writing target words according to Equation 3.4, as done standardly, or copying source words from $\mathbf{x}$, or $\texttt{copy}[x_i]$ according to the attention scores in Equation 3.8. A distribution is then computed over these actions using a `softmax` function and particular actions are chosen accordingly during training and decoding.

Algorithm 6 Neural Shortest Path Search

Input: Input $\mathbf{x}$, DAG $\mathcal{G}$, neural parameters Θ and non-linear function f, beam size l, source node b with init. score o.
Output: Shortest component path

1: $d[V[\mathcal{G}]] \leftarrow \infty, d[b] \leftarrow o, \pi[V[\mathcal{G}]] \leftarrow Nil$
2: $s[V[\mathcal{G}]] \leftarrow Nil$ ▷ Path state information
3: $s[b] \leftarrow$ `InitState()` ▷ Initialize source state
4: **for** each vertex $u \geq b \in V[\mathcal{G}]$ in sorted order **do**
5: **if** `isinf`$(d[u])$ **then** continue
6: $p \leftarrow s[u]$ ▷ Current state at node u, or $z_{<i}$
7: $L^1_{[l]} \leftarrow \underset{(v_1,\ldots,v_k)\in \texttt{Adj}[u]}{\arg\max}$ `softmax`$(f(\Theta, p, \mathbf{x}))$
8: **for** each vertex and label $(v, z) \in L$ **do**
9: `score` $\leftarrow -\log p_\Theta(z \mid p, \mathbf{x}) + d[u]$
10: **if** $d[v] >$ `score` **then**
11: $d[v] \leftarrow$ `score`$, \pi[v] \leftarrow u$
12: $s[v] \leftarrow$ `UpdateState`(p, z)
13: **return** FindPath$(\pi, |V|, b)$

Decoding and Learning The full decoding procedure is shown in Algorithm 6, where the differences with the standard SSSP are again shown in red. We change the data structure s to contain the decoder's RNN state at each node. We also modify the scoring (line 7, which uses Equation 3.4) to consider only the top l edges or translations at that point, as opposed to imposing a full search. When l is set to 1, for example, the procedure does a greedy search through the graph, whereas when l is large the procedure is closer to a full search.

In general terms, the decoder described above works like an ordinary neural decoder with the difference that each decision (i.e., new target-side word translation) is constrained (in line 7) by the transitions allowed in the underlying graph in order to ensure wellformedness of each component output. Standardly, we optimize these models using stochastic gradient descent with the objective of finding parameters Θ^* that minimize the negative conditional log-likelihood of the training dataset (see details in Section 2.3.2).

Algorithm 7 k-SSSP Decoding via Yen's Algorithm

Input: Input $\mathbf{x}$, DAG $\mathcal{G}$, SSSP method SP, number of paths K, translation mode θ, starting node b.
Output: K shortest paths A

$A[k] \leftarrow Nil$ ▷ Initialize the k-best list A
$B \leftarrow [\,]$ ▷ Initialize the k-best candidate list B
$A[0] \leftarrow \text{SP}(\mathbf{x}, \mathcal{G}, \theta, b)$ ▷ Find initial SSSP starting from b
for $k \in 1..K$ **do**
  for $i \in 0$ **to** $\text{LEN}(A[k-1]) - 1$ **do** ▷ Run through each node in recent SSSP
    `new_start` $\leftarrow A[k-1][i]$
    `root` $\leftarrow A[k-1][:i]$
    for each path $p \in A$ **do** ▷ Find all paths in A matching root
      if `root` $= p[0:i]$ **then**
        $\mathcal{G} \leftarrow \text{BLOCK}(\mathcal{G}, p[i], p[i+1])$
    `branching` $\leftarrow \text{SP}(\mathbf{x}, \mathcal{G}, \theta,$ `new_start`$)$ ▷ Find new SSSP from `new_start`
    `candidate` $\leftarrow$ `root` $+$ `branching`
    $B \leftarrow \text{HEAPPUSH}(B,$`candidate`$)$ ▷ Add `candidate` as a candidate shortest path
    $\mathcal{G} \leftarrow \text{UNBLOCK}(\mathcal{G})$
  $A[k] \leftarrow \text{HEAPPOP}(B)$ ▷ Add best candidate to A
return A

3.4.3 Monolingual vs. Polyglot Decoding

Our framework facilitates both monolingual and polyglot decoding. In the first case, the decoder requires a graph associated with the output semantic language (more details in next section) and a trained translation model. The latter case requires taking the union of all datasets and graphs (with artificial identifier tokens) for a collection of target datasets and training a single model over this global dataset. In this setting, we can then decode to a particular language using the language identifiers or decode without specifying the output language. The main focus in this chapter is investigating polyglot decoding, and in particular the effect of training models on multiple datasets when translating to individuals APIs or SP datasets.

When evaluating our models and building QA applications, it is important to be able to generate the k best translations. This can easily be done in our framework by applying standard k SSSP algorithms. We use an implementation of the algorithm of Yen (1971) shown in Algorithm 7. As detailed in Brander and Sinclair

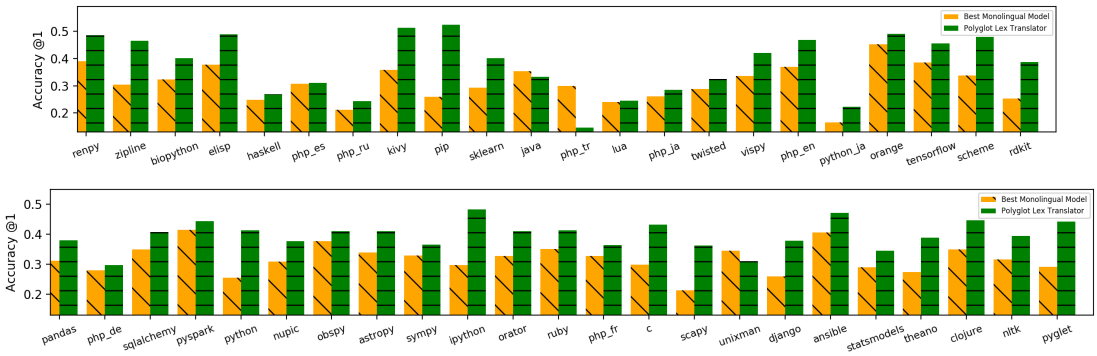


Figure 3.8: Test **Acc@1** for the best monolingual models (in yellow/left) compared with the best lexical polyglot model (green/right) across all 45 technical documentation datasets.

(1995) (see also Eppstein (2008)), this method is one of many k Shortest Path algorithms that works by finding deviating or branching paths from an initial SSSP (computed in Line 3). For each k starting on Line 4, the method then dissects the most recent shortest path and again uses the single shortest path method to find an alternative path from each point `new_node` that hasn't been observed in the current list A (as checked starting on line 8). See Appendix B for more details.

3.5 Experimental Setup

We experimented with two main types of resources: 45 API documentation datasets and two multilingual benchmark SP datasets. In the former case, our main objective is to test whether training polyglot models (shown as **polyglot** in Tables 3.1-3.2) on multiple datasets leads to an improvement when compared to training individual monolingual models (shown as **monolingual** in Tables 3.1-3.2). Experiments involving the latter datasets are meant to test the applicability of our general graph and polyglot method to related SP tasks, and are also used for comparison against our main technical documentation task.

3.5.1 Datasets

Technical API Docs The first dataset includes the Stdlib and Py27 datasets of Richardson and Kuhn (2017b,a) (from the last chapter), which are publicly available via Richardson (2017). Stdlib consists of short description and function signature pairs for 10 programming languages in 7 languages, and Py27 contains the same type of data for 27 popular Python projects in English mined from GitHub. We also built two new datasets from the Japanese translation of the Python 2.7 standard library, as well as the Lua stdlib documentation in a mixture of Russian, Portuguese, German, Spanish and English.

Taken together, these resources consist of 79,885 training pairs, and we experiment with training models on Stdlib and Py27 separately as well as together (shown as + **more** in Table 3.1). We use BPE subword encodings (Sennrich et al., 2015) of both input and output words to make the representations more similar and transliterated all datasets (excluding our Japanese datasets) to an 8-bit latin encoding. Graphs were built by concatenating all function representations into a single word list and compiling this list into a minimized `DAFSA`. For our global polyglot dataset, this resulted in a graph with 218,505 nodes, 313,288 edges, and 112,107 paths over an output vocabulary of 9,324 words.

Mixed GeoQuery and Sportscaster We run experiments on the GeoQuery 880 corpus using the splits from Andreas et al. (2013), which includes geography queries for English, Greek, Thai, and German paired with formal database queries, as well as a seed lexicon or *NP list* for each language. In addition to training models on each individual dataset, we also learn polyglot models trained on all datasets concatenated together. We also created a new mixed language test set that was built by replacing NPs in 803 test examples with one or more NPs from a different language using the NP lists mentioned above (see example in Figure 3.11). The goal in the last case is to test our model's ability to handle mixed language input. We also ran monolingual experiments on the English Sportscaster corpus, which contains human generated soccer commentary paired with symbolic meaning representation produced by a simulation of four games.

For GeoQuery graph construction, we built a single graph for all languages by

```
## entities
define NUMBERS [ 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 11 ];
define PURPLE [ purple NUMBERS ];
define PINK [ pink NUMBERS ];
define PPAIRS [ PURPLE PURPLE ];
define PKPAIRS [ PINK PINK ];
define TEAMS [ [ pink_ | purple_ ] team ];
define PLAYS [ free_ kick_ | kick_ in_ | goal_ | ... ];

## relations
regex [ pass@r [ PKPAIRS | PPAIRS ] ];
regex [ kick@r [ PURPLE | PINK ] ];
regex [ defense@r [ PURPLE | PINK ] ];
regex [ block@r [ PURPLE | PINK ] ];
regex [ turn@@ over@r [ PURPLE PINK | PINK PURPLE ] ];
regex [ bad@@ pass@r [ PURPLE PINK | PINK PURPLE ] ];
regex [ steal@r [ PURPLE | PINK ] from@l player@l ];
regex [ ball@@ stopped@r ball@l ];
regex [ play@@ mode@r PLAYS TEAMS ];
regex [ play@@ mode@r play_ on ];

union net
```

Figure 3.9: An example implementation of the Sportscaster language and graph expressed in the Xerox finite-state language.

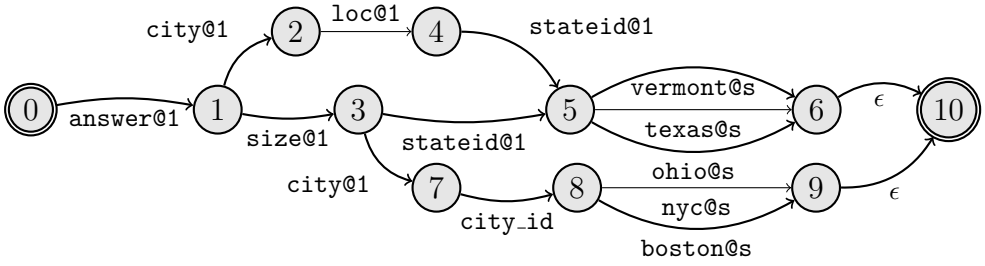


Figure 3.10: A DAG representation of the search space for a fragment of Geoquery.

		Method	Acc@1	Acc@10	MRR
stdlib	**mono.**	RK Trans + *rerank*	29.9	69.2	43.1
		Lexical SP	**33.2**	70.7	45.9
	poly.	Lexical SP + *more*	33.1	69.7	45.5
		Neural SP + *bias*	12.1	34.3	19.5
		Neural SP + *copy_bias*	13.9	36.5	21.5
py27	**mono.**	RK Trans + *rerank*	32.4	73.5	46.5
		Lexical SP	**41.3**	77.7	54.1
	poly.	Lexical SP + *more*	40.5	76.7	53.1
		Neural SP + *bias*	8.7	25.5	14.2
		Neural SP + *copy_bias*	9.0	26.9	15.1

Table 3.1: Test results on the Stdlib and Py27 tasks averaged over all datasets and compared against the best monolingual results from Richardson and Kuhn (2017b,a), or RK

extracting general rule templates from all representations in the dataset, and exploited additional information and patterns using the Geobase database and the semantic grammars used in Wong and Mooney (2006) (see Figure 3.10 for an example). This resulted in a graph with 2,419 nodes, 4,936 edges and 39,482 paths over an output vocabulary of 164. For Sportscaster, we directly translated the semantic grammar provided in Chen and Mooney (2008a) to a `DAFSA`, which resulted in a graph with 98 nodes, 86 edges and 830 paths (an example implementation of this graph is shown in Figure 3.9).

3.5.2 Evaluation

For the technical datasets, the goal is to see if our model generates correct signature representations from unobserved descriptions using exact match. We follow exactly the experimental setup and data splits from Richardson and Kuhn (2017b), and measure the accuracy at 1 (**Acc@1**), accuracy in top 10 (**Acc@10**), and **MRR**.

For the GeoQuery and Sportscaster experiments, the goal is to see if our models can generate correct meaning representations for unseen input. For GeoQuery, we follow Andreas et al. (2013) in evaluating extrinsically by checking that each representation evaluates to the same answer as the gold representation when executed against the Geobase database using a Prolog reasoner. For Sportscaster, we evaluate by exact match to a gold representation.

3.5.3 Implementation and Model Details

We use the Foma finite-state toolkit of (Hulden, 2009), which is an open source version of the well-known Xerox finite-state toolkit (Beesley and Karttunen, 2003), to construct all graphs used in our experiments. We also use the Cython version of Dynet (Neubig et al., 2017) to implement all the neural models (see Appendix B for more details).

In the results tables, we refer to the lexical and neural models introduced in Section 4 as *Lexical Shortest Path* and *Neural Shortest Path*, where models that use copying (*+ copy*) and lexical biasing (*+ bias*) are marked accordingly. We also experimented with adding a discriminative reranker to our lexical models (*+ rerank*), using the approach from Section 2.3.2, which uses additional lexical (e.g., word match and alignment) features and other phrase-level and syntax features. The goal here is to see if these additional (mostly non-word level) features help improve on the baseline lexical models.

3.6 Experimental Results and Discussion

Technical Documentation Results Table 3.1 shows the results for Stdlib and Py27. In the monolingual case, we compare against the best performing models in Richardson and Kuhn (2017b,a) (i.e., last chapter). As summarized in Figure 3.8, our experiments show that training polyglot models on multiple datasets can lead to large improvements over training individual models, especially on the Py27 datasets where using a polyglot model resulted in a nearly 9% average increase in accuracy @1. In both cases, however, the best performing lexical models are those trained only on the datasets they are evaluated on, as opposed to training on all datasets (i.e., *+ more*). This is surprising given that training on all datasets doubles the size of the training data, and shows that adding more data does not necessarily boost performance when the additional data is from another distribution.

The neural models are strongly outperformed by all other models both in the monolingual and polyglot case (only the latter results shown), even when lexical biasing is applied. While surprising, this is consistent with other studies on *low-resource* neural MT (Zoph et al., 2016; Östling and Tiedemann, 2017), where

		Method	Acc@1	Acc@10
Standard GeoQuery	monolingual	UBL Kwiatkowski et al. (2010)	74.2	–
		TreeTrans Jones et al. (2012b)	76.8	–
		nHT Susanto and Lu (2017)	**83.3**	–
		Lexical Shortest Path	68.6	92.4
		Lexical Shortest Path + *rerank*	74.2	94.1
		Neural Shortest Path	73.5	91.1
		Neural Shortest Path + *bias*	78.0	92.8
		Neural Shortest Path + *copy_bias*	77.8	92.1
	polyglot	Lexical Shortest Path	67.3	92.9
		Lexical Shortest Path + *rerank*	75.2	94.7
		Neural Shortest Path	78.0	91.4
		Neural Shortest Path + *bias*	78.9	91.7
		Neural Shortest Path + *copy_bias*	**79.6**	91.9
Mixed	poly.	Best Monolingual Model	4.2	18.2
		Lexical Shortest Path + *rerank*	71.1	94.3
		Neural Shortest Path + *copy_bias*	**75.2**	90.0
Sportscaster	mono.	PCFG Börschinger et al. (2011)	74.2	–
		wo-PCFG Börschinger et al. (2011)	**86.0**	–
		Lexical Shortest Path	40.3	86.8
		Lexical Shortest Path + *rerank*	70.3	90.2
		Neural Shortest Path	81.9	94.8
		Neural Shortest Path + *bias*	83.4	93.9
		Neural Shortest Path + *copy_bias*	83.3	90.5

Table 3.2: Test results for the standard (above) and mixed (middle) GeoQuery tasks averaged over all languages, and results for the English Sportscaster task (below).

datasets of comparable size to ours (e.g., 1 million tokens or less) typically fail against classical SMT models. This result has also been found in relation to neural AMR semantic parsing, where similar issues of sparsity are encountered (Peng et al., 2017). Even by doubling the amount of training data by training on all datasets (results not shown), this did not improve the accuracy, suggesting that much more data is needed (more discussion below).

Beyond increases in accuracy, our polyglot models support zero-shot translation as shown in Figure 3.11, which can be used for translating between unobserved language pairs (e.g., (*es*,`Clojure`), (*ru*,`Haskell`) as shown in 1-2), or for finding related functionality across different software projects (as shown in 3). These results were obtained by running our decoder model without specifying the output language. We note, however, that the decoder can be constrained to selectively translate to any specific programming language or project (e.g., in a QA setting). Future work will further investigate the decoder's polyglot capabilities, which is

1.	Source API (stdlib): (***es*, PHP**)	**Input**: Devuelve el mensaje asociado al objeto lanzado.
Output	Language: **PHP**	Function Translation: `public string Throwable::getMessage ( void )`
Output	Language: **Java**	Function Translation: `public String lang.getMessage( void )`
Output	Language: **Clojure**	Function Translation: `(tools.logging.fatal throwable message & more)`
2.	Source API (stdlib): (***ru*, PHP**)	**Input**: конвертирует строку из формата UTF-32 в формат UTF-16.
Output	Language: **PHP**	Function Translation: `string PDF_utf32_to_utf16 ( ... )`
Output	Language: **Ruby**	Function Translation: `String#toutf16 => string`
Output	Language: **Haskell**	Function Translation: `Encoding.encodeUtf16LE :: Text -> ByteString`
3.	Source API (py): (***en*, stats)**	**Input:** Compute the Moore-Penrose pseudo-inverse of a matrix.
Output	Project: **sympy**	Function Translation: `matrices.matrix.base.pinv_solve( B, ... )`
Output	Project: **sklearn**	Function Translation: `utils.pinvh( a, cond=None,rcond=None,... )`
Output	Project: **stats**	Function Translation: `tools.pinv2( a,cond=None,rcond=None )`
4.	Mixed GeoQuery (de/gr)	**Input**: Wie hoch liegt der höchstgelegene punkt in Αλαμπάμα?
	Logical Form Translation: `answer(elevation_1(highest(place(loc_2(stateid('alabama'))))))`	

Figure 3.11: Examples of zero-shot translation when running in polyglot mode (1-3, function representations shown in a conventionalized format), and mixed language parsing (4).

currently hard to evaluate since we do not have an annotated set of function equivalences between different APIs.

Semantic Parsing Results SP results are summarized in Table 2. In contrast, the neural models, especially those with biasing and copying, strongly outperform all other models and are competitive with related work. In the GeoQuery case, we compare against two classic grammar-based models, UBL and TreeTrans, as well as a feature rich, neural hybrid tree model (nHT). We also see that the polyglot Geo achieves the best performance, demonstrating that training on multiple datasets helps in this domain as well. In the Sportscaster case we compare against two PCFG learning approaches, where the second model (wo-PCFG) involves a grammar with complex word-order constraints.

The advantage of training a polyglot model is shown on the results related to mixed language parsing (i.e., the middle set of results). Here we compared against the best performing monolingual English model (**Best Mono. Model**), which does not have a way to deal with multilingual NPs. We also find the neural model to be more robust than the lexical models with reranking. One conclusion therefore is that polyglot models can effectively facilitate mixed language decoding.

While the lexical models overall perform poorly on both tasks, the weakness of

this model is particularly acute in the Sportscaster case. We found that mistakes are largely related to the ordering of arguments, which these lexical (unigram) models are blind to. For example, the target LF translation for the sentence *purple player 2 kicks to purple 5* is the following (more details about this dataset are provided in the next chapter):

```
pass(purple2,purple5)
```

We found that in such cases the lexical models often confuse the ordering of arguments like `purple2` and `purple5`. That these models still perform reasonably well on the GeoQuery task shows that such ordering issues are less of a factor in this domain (and perhaps that the underlying sequence prediction problem is less difficult when the general syntax of the output language is known).

Algorithmic Analysis: RankDecoder versus Lexical SSSP As already discussed, the rank decoding strategy (or RankDecoder, as defined in Algorithm 2) that we started with in the last chapter has the disadvantage of binding the search complexity to the size of the output target language $\mathcal{C}$, and runs in time $O(|\mathcal{C}| \log |\mathcal{C}|)$. The lexical SSSP (LexDecoder) decoder developed here (which relies on the same underlying SMT model) improves on this by using a more general DAG k-SSSP search strategy that runs in time $O(k\,|V|\,(|V| + |E|))$ (for more theoretical analysis, see Appendix B), which in theory scales better to larger languages. Despite this, the lexical SSSP is an approximate search strategy (unlike the RankDecoder), since it ignores the normalization constant $|\mathcal{A}|$, and might not scale well in practice for large graphs $\mathcal{G}$. Figure 3.12 shows the average decoding times for these two different strategies as a function of the beam k for our largest datasets (recall that the RankDecoder ranks all signatures, hence the flat line), along with a comparison of accuracy. Here we see that for individual datasets (i.e., Figure 3.12.a-d) the LexDecoder does not scale well for larger k relative to the RankDecoder, though in terms of matching accuracy, only a small k seems needed in most cases. For larger datasets (Figure 3.12e-f), however, the LexDecoder is much more efficient, even despite requiring a larger beam k (e.g., in the case of GeoQuery) to achieve comparable performance.

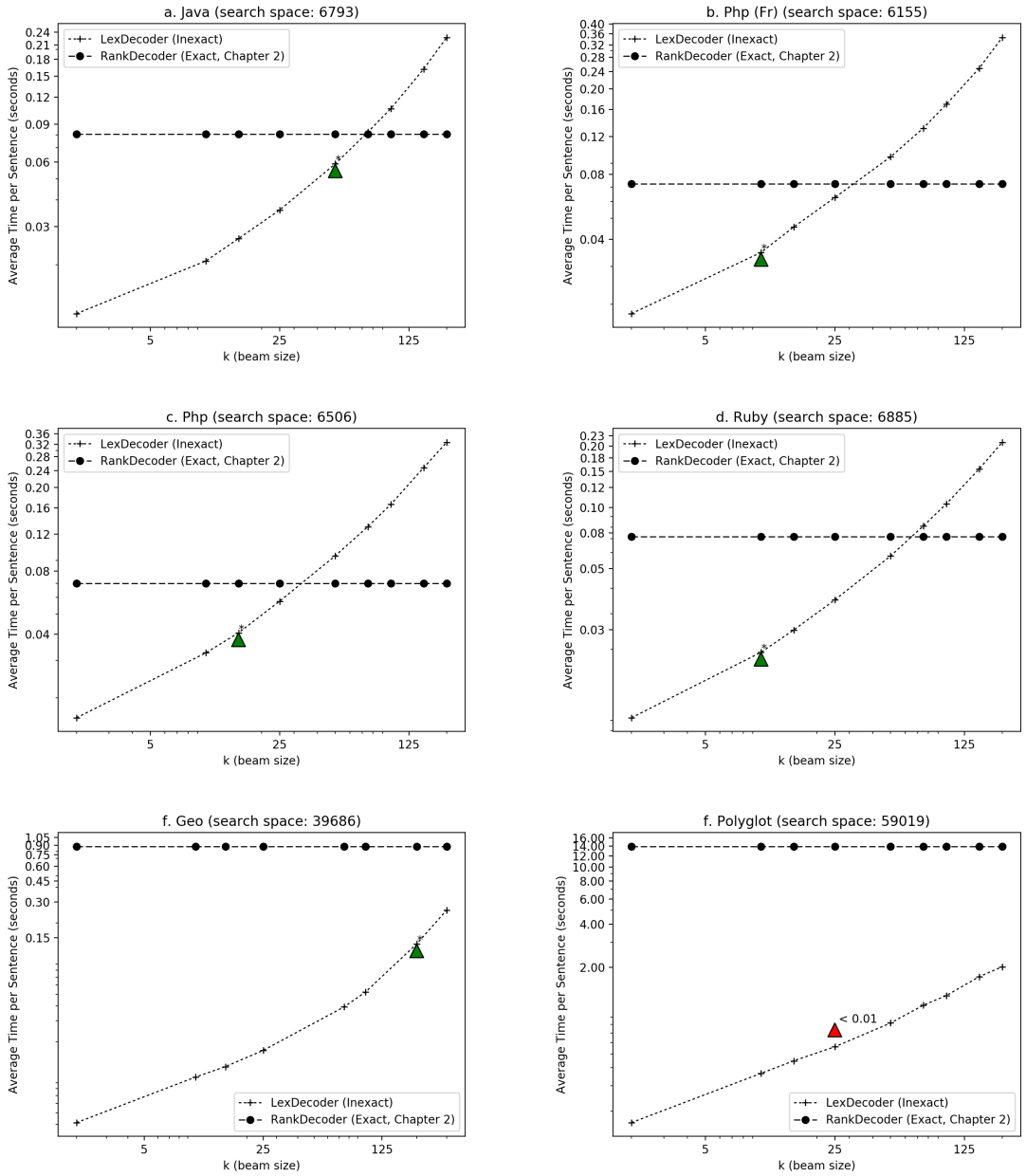


Figure 3.12: Average decode time as a function of k for rank decoders (RankDecode) and lexical SSSP decoders (LexDecoder) (log-log scale, triangles show point at which the accuracy@1 either matches (green) or is comparable (red) with rank decoder).

In terms of system building, one takeaway is that the RankDecoder appears to be sufficiently efficient for small API projects, though using our graph model can lead to considerable speed improvements for small sizes of k (which generally appear to achieve comparable accuracy). For larger collections of APIs and semantic languages, however, the RankDecoder can become prohibitively slow, and the use of our graph decoder becomes essential. Therefore, the choice of either method should be tailored to the nature and size of the target API.

Discussion Having results across related SP tasks allows us to reflect on the nature of the main technical documentation task. Consistent with recent findings (Dong and Lapata, 2016), we show that relatively simple neural sequence models are competitive with, and in some cases outperform, traditional grammar-based methods on benchmark SP tasks. However, we find that using the exact same neural models in the technical documentation domain does not deliver the same positive results. This asymmetry appears related to the fact that in the technical docs, each code representation being predicted is almost entirely unobserved and unique (owing to the fact the developers do not often define redundant factors with identical parameters), which makes generalization hard for complex sequence models. In contrast, the representations being learned in GeoQuery are much more redundant (see Table 2.1 for a comparison), to the extent that virtually all testing representations are observed in the training phase.

For this reason, we believe our datasets provide new challenges for neural-based SP and serve as a cautionary tale about the applicability of these models to lower-resource tasks. One natural question to ask is the following: how much additional data is needed to improve the neural models on this task? In looking at the results, we observe that the one place where the neural models do well is with the PHP datasets, which are translated in 7 languages (hence, we might say that our PHP dataset is 7 times larger than all other datasets). Despite being competitive, however, the polyglot neural models are still outperformed in all cases by the SMT models, showing that even with 7 times more data, the system still might not be able to learn effectively. Rather than focusing on building more data, one alternative approach might be to develop sequence models that exploit more abstract structure in the code representations, such as type sequences or abstract

sketches (Dong and Lapata, 2018), which can be *coupled* together with simpler lexical models to predict code words (as looked at in Herzig and Berant (2018)).

Turning to our initial research question, however, we see that training on multiple datasets (i.e., polyglot modeling) can be an effective technique for learning more robust translation models (assuming that the appropriate translation model is used) and for learning across multiple domains. Beyond improving translation quality, one main benefit of the polyglot approach is its ability to support mixed language decoding, which we introduce as a new evaluation task in the GeoQuery domain, as well as translating to multiple programming languages. We find these new types of evaluation to be more revealing than only considering translation accuracy, since they reveal fundamental differences between what different models (which might achieve comparable accuracy) are capable of doing. We hope that our work helps to motivate more diverse evaluations of this type for SP.

In terms of our second research question about building decoding models that support zero-shot translation, we find that doing zero-shot translation (and other types of polyglot decoding) follows rather straightforwardly by our graph-based decoder, which was initially developed as an improvement over the constrained decoding methods used in the last chapter. In general, our basic idea of constraining translation search using graphs has close similarity with recent methods in neural SP on using decoders that generate grammar representations (Krishnamurthy et al., 2017; Cheng et al., 2017; Yin and Neubig, 2017). While our graph approach is limited to acyclic graphs, largely due to the nature of the component representations, one could extend this basic method to more expressive graphs (e.g., cyclic graphs or tree structure hypergraphs) by simply employing a more expressive SSSP search. We believe, therefore, that our shortest path decoding strategy could serve as a more general search framework for constrained decoding.

3.7 Conclusions

In this chapter, we looked at learning from multiple API libraries and datasets in the context of learning to translate text to code representations and other SP tasks. The central intuition is that by building single models over multiple datasets with shared parameters (i.e., polyglot models), such models are able to capture

certain redundancies across different datasets, and hence facilitate more robust translation and SP. To support polyglot modeling of this type, we introduced a novel graph-based decoding and search framework and experimented with various SMT and neural MT models that work in this framework.

In conclusion, we found polyglot modeling to be a useful technique for improving translation and for transfer learning across different (sometimes disparate) domains and languages. Using this method, we achieved large improvements on the 43 (+2) technical documentation tasks first introduced in the last chapter, and also demonstrated the usefulness of this technique on two additional benchmark SP tasks, on which we achieved results competitive with the state-of-the-art. In order to highlight the benefit of polyglot modeling, we also introduced a novel mixed language SP task and GeoQuery test set, on which our polyglot models achieved large improvements over training monolingual (language specific) models.

These positive results, however, come with certain caveats. Building on a theme from the last chapter, we found that the technical documentation datasets provide new challenges for SP largely due to their large scope and sparsity, and that the positive results mentioned above are dependent on the particular models being employed. By experimenting with additional benchmark SP tasks, we were able to directly compare the performance of recent state-of-the-art neural sequence models on these benchmark tasks to results on our technical documentation task. We found that such neural models fail to achieve comparable results to simpler SMT models, which, we believe, highlights the limitations of these models for more general SP tasks and the need for developing more robust neural SP architectures.

Moving ahead, we see a lot of potential for using polyglot modeling to do more complex types of transfer learning (e.g., data augmentation and modeling paraphrasing and entailment across domains), and for doing knowledge acquisition in the source code domain (which is a topic that we return to in Chapter 5).

4 Learning from Entailment for Semantic Parsing

"The basic aim of semantics is to characterize the notion of a true sentence (under a given interpretation) and of entailment."

– Richard Montague, *Universal Grammar (1970)*

4.1 Modeling Entailment for Semantic Parsing

4.1.1 The Idea

Throughout this thesis, we have treated semantic parsing as primarily a translation problem, which we have studied independently of the other subtasks (i.e., knowledge representation and symbolic reasoning) associated with the general NLU program outlined in Chapter 1. It is worth bearing in mind, however, that the ultimate goal, as described in the quote above by Montague, is to generate formal meaning representations that capture facts about truth and entailment and facilitate deep symbolic reasoning. In this chapter, we examine the following question: do the formal representations being learned for semantic parsing actually help us to model entailment, and if not, how can we learn representations that do?

To illustrate this idea, Figure 4.1 shows a variant of the pipeline model introduced in Chapter 1 that includes the following sentence (in red) that is *logically entailed* by the first sentence (for details about entailment, see Appendix C):

Find some sample that contains a major element. (4.1)

A consequence of this logical entailment is that the denotation of the second sentence (i.e., the set of answers that make this sentence true) should always be a subset of the denotation of the first sentence, regardless of the target dataset or knowledge source being used. Linguists in the Montague tradition have long used

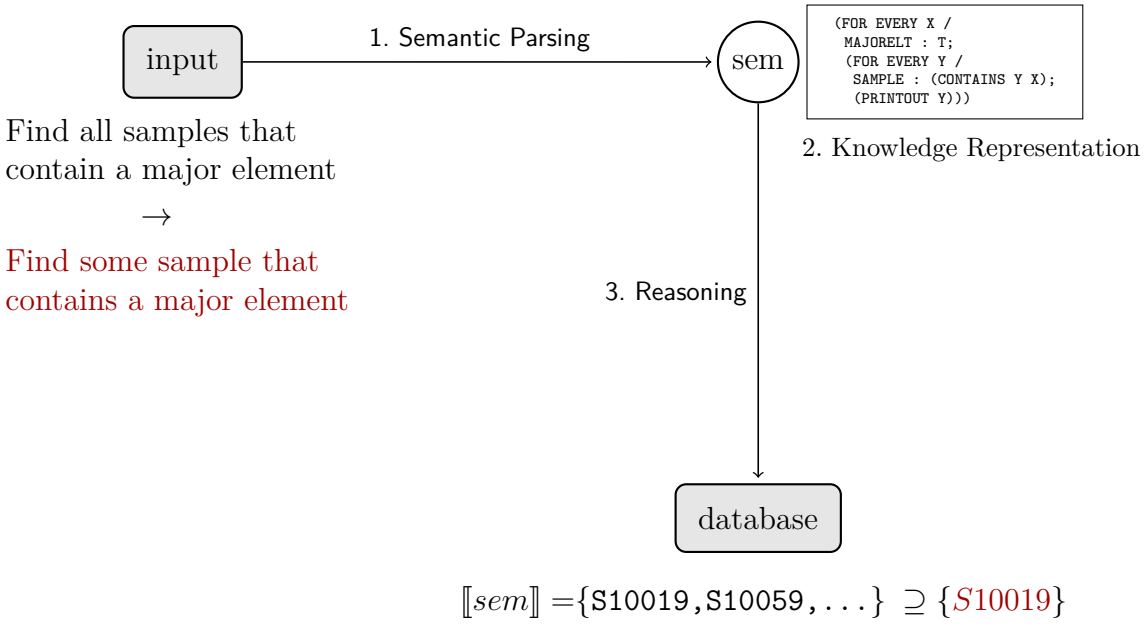


Figure 4.1: The global NLU picture with entailment.

judgements about entailment as the main tool for motivating and evaluating different theories of semantics. Using an analogy with programming and software, we can think about tests of entailment in semantics as a kind of *unit test* for system development, as described below:

- **Entailment as a Unit Test**: For a set of target sentences, check that our semantic model accounts for particular entailment patterns observed between pairs of sentences; modify our model when such tests fail.

The question investigated here is: what happens when we subject our semantic parsers to such a unit test? In doing this, we adopt the loose definition of entailment used in the recognizing textual entailment challenges (RTE), where entailment is defined in terms of the following task (Dagan et al., 2005): given a **text** t and **hypothesis** h, determine if h is entailed by t where say that t entails h if a human reading t would typically infer that h is most likely true. Figure 4.2 shows example sentence pairs and logical forms (or LFs, generated by a semantic parser) from the Sportscaster corpus (Chen and Mooney, 2008a) already encountered in

			Entailments	
	Text t and gold `LF`	**Hypothesis** h and gold `LF`	**Human**	**Naïve**
1.	Pink 3 quickly kicks to Pink 7 `pass(pink3,pink7)`	Pink 3 kicks over to Pink 7 `pass(pink3,pink7)`	t `(entail)` h h `(uncertain)` t	`entail`
2.	Purple 10 kicks the ball `kick(purple10)`	Purple 10 shoots for the goal `kick(purple10)`	t `(uncertain)` h h `(entail)` t	`entail`
3.	Pink 10 kicks the ball `kick(pick10)`	Pink 10 passes over to Pink 7 `pass(pink10,pink7)`	t `(uncertain)` h h `(entail)` t	`contr.`
4.	Pink 7 makes a long kick `kick(purple7)`	Purple team scores another goal `playmode(goal_l)`	t `(uncertain)` h h `(uncertain)` t	`contr.`

Figure 4.2: Example sentence pairs and entailments in the Sportscaster domain.

Chapter 3. Each example is marked with an entailment judgement provided by humans in both the $t \rightarrow h$ and $h \rightarrow t$ directions. For example, in Figure 4.2-1, we can paraphrase the entailment from $t \rightarrow h$ in the following way:

> In all scenarios (e.g., possible game events) in which 'pink 3 quickly kicks to pink 7' is true, it is always simultaneously true (or nearly always true) that 'pink 3 kicks over to pink 7'

Subjecting our semantic parsers to an RTE test involves seeing if the semantic `LF` representations being generated can be used to derive and identify such correct entailments (i.e., entailments that are consistent with human judgements).

4.1.2 Yet Another Resource Problem!

The problem with the corpus `LF`s in Figure 4.2, however, is that while they capture the general events being discussed, they often fail to capture other aspects of meaning. Here, the **naïve** judgement is the entailment generated by comparing the `LF`s associated with t and h (i.e., by assigning `entail` when the `LF`s match, and `contradict` when they mismatch), which captures the full inferential power of the target `LF`s. In several cases, the naïve inferences result in judgements that are inconsistent with the human judgements. Therefore, some of the semantic representations provided in the corpus fail to pass the test described above.

In considering these examples, we identify the following two issues:

1. **Imprecise Corpus Representations**: The corpus representations fail to account for certain aspects of meaning. For example, the first two sentences in Figure 4.2-1 map to the same formal meaning representation (i.e., `pass(pink3,pink7)`) despite having slightly different semantics and divergent entailment patterns. This shift in meaning is related to the adverbial modifier *quickly*, which is not explicitly analyzed in the target representation. The same is true for the modifier *long* in example 4, and for all other forms of modification. For a semantic parser or generator trained on this data, both sentences in 1 are treated as having an identical meaning.

   As shown in the example 2, other representations fail to capture important sense distinctions, such as the difference between the two senses of the `kick` relation. While *shooting for the goal* in general entails *kicking*, such an entailment does not hold in the reverse direction. Without making this distinction explicit at the representation level, such inferences and distinctions cannot be made.

2. **Missing Domain Knowledge** Since the logical representations are not based on an underlying logical theory or domain ontology, semantic relations between different symbols are not known. For example, computing the entailments in example 3 requires knowing that in general, a `pass` event entails or implies a `kick` event (i.e., the set of things *kicking* at a given moment includes the set of things *passing*). Other such factoids are involved in reasoning about the sentences in example 4: `purple7` is part of the `purple team`, and a `score` event usually entails a `kick` event (but not conversely).

The more general resource problem involved here can be described in the following way: while we have a sufficient amount of parallel data for training a semantic parser in a given domain (thus solving the initial resource problems discussed in Sections 2.1.2 and 3.1.2), the gold `LF`s provided in the corpus are deficient and not able to capture the full range of NLU phenomena. Recalling our setup from Chapter 1, as shown again in Figure 4.3, this issue also touches on the shortcomings of how we evaluate our semantic parsing models.

One common way to deal with such resource problems is to re-annotate the corpus representations and the relevant background knowledge (Toledo et al., 2013).

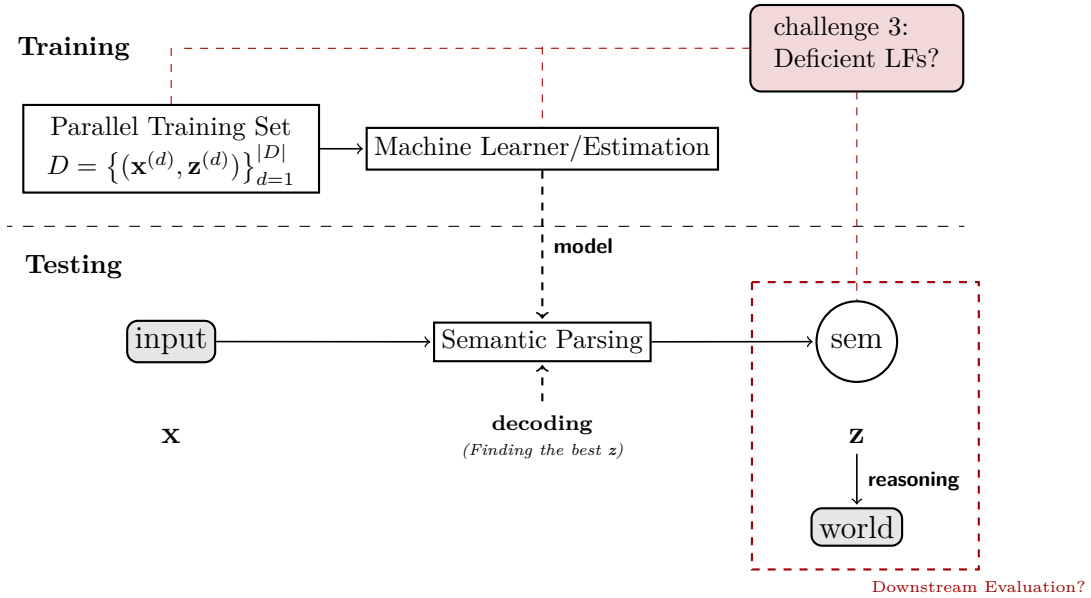


Figure 4.3: The standard semantic parsing setup and the third resource challenge (from Section 1.3).

We instead investigate whether this missing information can be learned, in particular by using example entailment judgements as a weak form of training supervision, which is a new learning framework that we call *learning from entailment*. Similar to the idea of *learning from denotation* in semantic parsing (Clarke et al., 2010; Liang et al., 2013; Berant et al., 2013), the intuition behind learning from entailment is that entailments give general information about denotations (i.e., the set of possible scenarios associated with entities and events), and that asymmetries in entailment judgements can be used for finding holes in the target representations and learning better representations. For example, given the mismatch in the entailments in Figure 4.2-1, one can infer that **t** has more specific information than **h** (or that its denotation is a subset of the denotations of h), which then requires learning a model that can identify this additional information and ultimately derive the semantics of this missing information.

To experiment with this idea, we introduce a new semantic parsing model that learns jointly using structured meaning representations (as done in previous ap-

proaches) and raw textual inference judgements between random pairs of sentences. In order to learn and model entailment phenomena, our model integrates natural logic (symbolic) reasoning (MacCartney and Manning, 2009) directly into our semantic learner. We perform experiments on the Sportscaster corpus (Chen and Mooney, 2008a), which we extend by annotating pairs of training sentences in the original dataset with inference judgements. On a new RTE-style inference task based on this extended dataset, we achieve an accuracy of 73%, which is an improvement of 13 percentage points over a strong baselines. As a separate result, part of our approach outperforms previously published results (from around 89% accuracy to 96%) on the original Sportscaster semantic parsing task.

4.2 Related Work

As reviewed in the last several chapters, work in semantic parsing has focused on learning semantic parsers from parallel data, often in the form of raw collections of text-meaning pairs. The earliest attempts (Kate et al., 2005; Wong and Mooney, 2006; Zettlemoyer and Collins, 2009) focused on learning to map natural language questions to simple database queries for database retrieval using collections of target questions and formal queries (e.g., in the GeoQuery domain studied in the last chapter). A more recent focus has been on learning representations using weaker forms of supervision that require minimal amounts of manual annotation effort (Clarke et al., 2010; Liang et al., 2011; Krishnamurthy and Mitchell, 2012; Artzi and Zettlemoyer, 2013; Berant et al., 2013), which includes work on learning from denotation (see Liang and Potts (2015); Liang (2016)).

Most work done on learning from denotation, and indeed in semantic parser induction more generally, has centered around question-answering (QA) applications. For example, Liang et al. (2011); Berant et al. (2013) train semantic parsers in QA domains using the denotations (or answers, represented as discrete symbolic entities) of each question as the primary supervision. One can regard this approach as the simplest form of learning from entailment; given a fixed database (or a model of all known scenarios) and symbolic representations of all answers, the aim is to learn a semantic parser given information that that each question is entailed by its associated answer. Under this scenario, however, entailment is lim-

ited to entailments between questions and simple answers (often existential values of some kind), and does not involve entailments that involve abstract relations between generic events and predicates, as we consider in this work. In general, entailment and symbolic reasoning has played a marginal role in existing work in semantic parsing, perhaps largely due to the primary focus on simple QA.

One inherent difficulty in modeling entailment (especially in RTE settings) and learning more complex semantic parsing representations is the need for considerable amounts of background knowledge (LoBue and Yates, 2011; Clark, 2018). Attempts to integrate more general knowledge into semantic parsing pipelines have often involved additional hand-engineering or external lexical resources (Wang et al., 2014; Tian et al., 2014; Beltagy et al., 2014). As discussed above, our approach looks at learning background knowledge indirectly from scratch by optimizing our models to predict the correct entailments, which to our knowledge has not been done before in semantic parsing work.

4.3 Problem Description and Approach

In this section, we given a high-level description of the original Sportscaster semantic parsing task and our approach to learning from entailment. While we define each task separately, we train our semantic parsing models jointly and in an end-to-end fashion using a grammar-based approach. A key technical innovation in our approach is the integration of formal symbolic reasoning into our semantic parsing model, which we describe in the next section and sketch out in Section 4.3.2.

4.3.1 The Sportscaster Task

Figure 4.4 shows a training example from the original Sportscaster corpus, consisting of a text about a sports event $\mathbf{x}$ paired with a set of formal meaning representations $\mathbf{Z}$. In this case, each text was collected by having human participants watch a 2-d simulation of several Robocup soccer league games (Kitano et al., 1997) and comment on events in the game. Rather than hand annotating the associated logical forms, sentences were paired with symbolic renderings of the underlying simulator actions that occurred around the time of each comment.

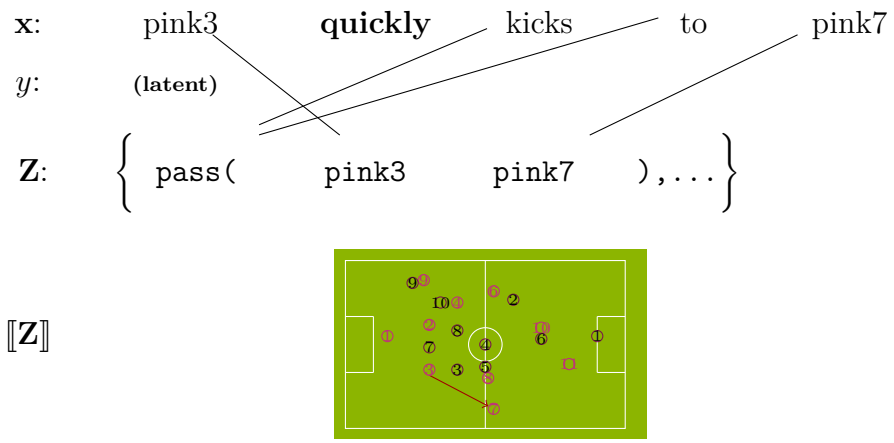


Figure 4.4: The original Sportscaster training setup.

These representations therefore serve as a proxy for the denotation of the event context and the individual events (shown as $[\![\mathbf{Z}]\!]$).

The goal is to learn a semantic parser `sp` given a training set D consisting of example sports descriptions and `LF`s, $D = \{(\mathbf{x}^{(d)}, \mathbf{Z}^{(d)})\}_{d=1}^{|D|}$, that can translate unseen descriptions to the correct `LF`s, as expressed below:

$$\texttt{sp} : \text{description} \rightarrow \texttt{LF}\ (\mathbf{z}) \tag{4.2}$$

In contrast to other work on learning from logical forms (e.g., in Chapter 2), the learning problem in this case is harder since the training data contains sets of possible `LF`s, as opposed to only gold `LF`s, which requires learning from *ambiguous supervision* (Mooney, 2008). The underlying idea is that these ambiguous contexts simulate the broader perceptual context associated with each comment, and hence provide a more realistic learning scenario.

The provided `LF`s (see examples in Figure 4.2) are expressed as atomic formulas in predicate logic defined over a small set of domain-specific predicates (e.g., `kick`, `pass`, `block`) and terms (e.g., `pink_team`, `pink1`, `purple11`). While our primary semantic parsing model generates exactly the representations provided in the original corpus, we reinterpret the semantics of these formulas in a way that it

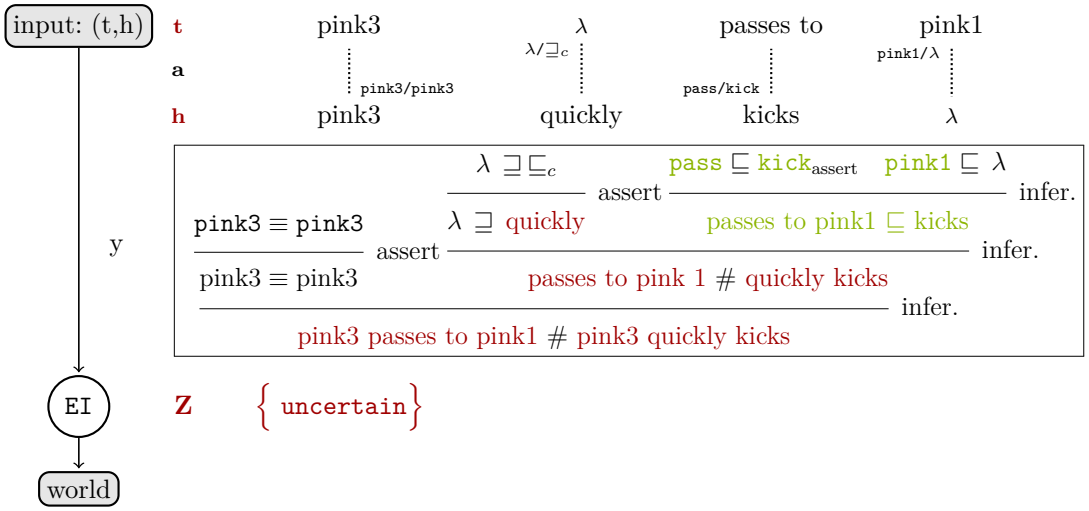


Figure 4.5: An example of learning from entailment.

makes it easier to model entailment. Specifically, terms are interpreted as separate predicates and the original event predicates are interpreted in a Neo-Davidsonian fashion (Parsons, 1990), as in the following example:

$$[\![\texttt{pass(pink3,pink7)}]\!] = \exists e.\exists x.\exists y.\texttt{pass}(e) \wedge \texttt{pink3}(x) \wedge \texttt{pink7}(y) \wedge \texttt{arg1}(x,e) \wedge \texttt{arg2}(y,e)$$

where the terms `pink3` and `pink7` are treated as separate predicates (which makes it easier to model abstract relationships such as $[\![\texttt{pink1}]\!] \subset [\![\texttt{pink\_team}]\!]$) and event predicates and predicate argument information apply over event variables e (in the first case, making it easier to model relationships such as $[\![\texttt{pass}]\!] \subset [\![\texttt{kick}]\!]$).

4.3.2 Learning from Entailment

The problem with the approach described above, as discussed in Section 4.1.2, is that the representations being learned do not always capture the types of information needed for modeling entailment. The general idea of learning from entailment is to extend a given semantic parsing dataset D with pairs of training sentences

annotated with inference judgements (as shown in Figure 4.2). While training an ordinary semantic parser `sp`, we then use such pairs to train a model `infer` that can generate certain types of entailments from example pairs of descriptions, as shown below:

$$\texttt{infer} : \Big(\text{description}_1 = \mathbf{t}, \text{description}_2 = \mathbf{h}\Big) \rightarrow \texttt{entailment}\ (\mathbf{z}) \tag{4.3}$$

where entailments can be of the following three types (Cooper et al., 1996; Bentivogli et al., 2011): {`entail`, `contradict`, `uncertain/compatible`}. The approach pursued here involves integrating a logical reasoning system into our semantic parser that can reason about the target symbols being learned and prove theorems about the target entailments. The key idea is that the resulting proofs reveal distinctions not captured in the original representations, and can be used to improve the semantic parser's internal representations and acquire knowledge.

An illustration of this is shown in Figure 4.5, where the input consists of a text $\mathbf{t}$ and hypothesis $\mathbf{h}$, and and a set of entailment judgements $\mathbf{Z}$ (in this case, a single `uncertain` judgement, which we represent using the variable `z`, as with `LF`s). y shows an example proof, or explanation, of how the model arrives at an `uncertain` inference based on a set of local inferences about relationships between aligned (via a) parts of $\mathbf{t}$ and $\mathbf{h}$. For example, the model reasons that *pass to pink1* entails *kicks* (based on some *assertion* or axiom between `pass` and `kick`), whereas uncertainty is introduced with the modifier *quickly* in the hypothesis and *passes to pink1* and *quickly kicks*; this uncertainty then propagates up the proof using generic inference rules *infer* defined in the model (to be described in Section 4.4.2). Since example proofs are not provided at training time, the learning problem is to find the correct proofs within a large latent space of possible proofs.

This particular proof gives rise to several new assertions or facts: the `pass` symbol is found to forward entail or imply (shown using the set inclusion symbol $\sqsubseteq$) the `kick` symbol. The adverbial modifier, which is previously unanalyzed, is treated as an entailing modifier $\sqsubseteq_c$, which results in a reverse entailment or implication (shown using the symbol $\sqsupseteq$) when inserted (or substituted for the empty symbol λ) on the hypothesis side. The first fact can be used for building a domain theory, and the second for assigning more precise labels to modifiers in the semantic parser.

In the latter case, we might assign the following *improved* representation to the input *pink3 quickly kicks* (using the semantics described in the previous section):

$$\exists e.\exists x.\texttt{kick}(e) \wedge \texttt{quickly}(e) \wedge \texttt{pink3}(x) \wedge \texttt{arg1}(x, e)$$

in which we have a new predicate `quickly` derived from its use as a forward entailing modifier in the example proof.

Computing entailments in our approach is specifically driven by learning the correct semantic assertions between primitive domain symbols, as well as the semantic effect of deleting/inserting symbols. We focus on learning the following very broad types of linguistic inferences (Fyodorov et al., 2003):

- **construction-based**: inferences generated from specific (syntactic) constructions or lexical items in the language
- **lexical-based**: inferences generated between words or primitive concepts due to their inherent lexical meaning

Construction-based inferences are inferences related to modifier constructions: *quickly*(`pass`) $\sqsubseteq$ `pass`, `goal` $\sqsupseteq$ *nice*(`goal`), *gets_a*(`free kick`) $\equiv$ (equivalence) `free kick`, where the entailments relate to default properties of particular modifiers when they are added or dropped. Lexical-based inferences relate to general inferences and implications between primitive semantic symbols or concepts: `kick` $\sqsupseteq$ `score`, `pass` $\sqsubseteq$ `kick`, and `pink1` $\sqsubseteq$ `pink team`.

4.4 Grammar-based Semantic Parsing

To model `sp` and `infer`, we use a grammar-based approach based on probabilistic context-free grammars (PCFG). In both cases, the target model assigns to each input a tree structured representation corresponding either to an `LF` representation or an entailment **z**. Grammar models build such structures using a finite set of (probabilistic) rewrite rules, which are created via a rule extraction process defined over the target parallel data described in the previous sections (as illustrated in Figure 4.6). In this section, we discuss the general PCFG formalism and explain its use in semantic parsing, then describe our rule extraction procedure (Sec-

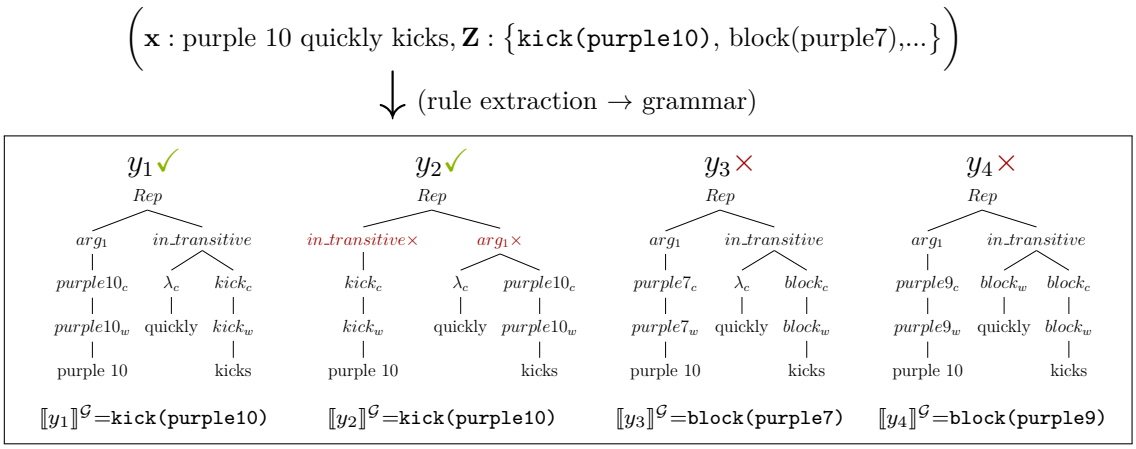


Figure 4.6: Semantic grammar rule extraction and example derivations.

tion 4.4.1) and the integration of logical reasoning into this model (Section 4.4.2). In Section 4.4.3 we finish by describing how we estimate our models from parallel data using a simple EM bootstrapping approach.

Modeling Preliminaries: Translating with PCFGs

Formally, a PCFG defines a 5-tuple $\mathcal{G}_\theta = (\Sigma, N, S, R, \theta)$ consisting of a set of terminal (i.e., source language) symbols Σ, a set of non-terminal grammar symbols N, a start symbol $S \in N$, a set of rewrite rules $R = \{N \to \beta \mid \beta \in (N \cup \Sigma)^*\}$ and a parameter vector $\theta \in \mathbb{R}^{|R|}$ (without θ, this defines a CFG). Using θ, each rule $N \to \beta$ (consisting of a left hand side (lhs) N and a right hand side (rhs) β) is assigned a score $\theta_{N\to\alpha}$ subject to the following constraints (where R_N is used to denote the set of rules from R that share the same lhs N):

$$\forall N \to \beta \quad 0 \leq \theta_{N\to\beta} \leq 1$$
$$\forall R_N \sum_{(N\to\alpha)\in R_N} \theta_{N\to\alpha} = 1$$

A derivation y over an input $\mathbf{x}$ is any application of rules that results in a tree rooted by S such that yield of the tree (i.e., the sequence of terminal nodes in the tree) is equal to $\mathbf{x}$. For example, Figure 4.6 shows an example derivation y for the

sentence *purple 10 quickly kicks* (shown below in a standard Lisp format):

```
(Rep
 (arg1 (purple10c (purple10w purple 10)))
 (in_transitive (λc quickly)
                (kickc
                 (kickw kicks))))
```

where rules include $\{\texttt{Rep} \rightarrow \texttt{arg}_1\texttt{in\_transitive},\ \texttt{arg}_1 \rightarrow \texttt{purple}_c, ...\} \subseteq R$ with the start node `Rep`, and the yield of the derivation is the left-to-right sequence of terminal symbols *purple 10 quickly kicks* (i.e., the input sentence). Imagining that probabilities are associated with rules, the score of this derivation y is computed as a product over the individual rule probabilities $N_j \rightarrow \beta_j$ in that derivation:

$$p_\theta(y) = \prod_i^{|y|} \theta_{N_i \rightarrow \beta_i} \tag{4.4}$$

As a generative model, PCFGs can be used to model the joint distribution $p(\mathbf{x}, y)$, which allows us to compute the probability of a given input $\mathbf{x}$ by marginalizing over all derivations over $\mathbf{x}$, or $\mathcal{Y}_\mathbf{x}$ (where computing each joint probability reduces to computing the probability of each derivation):

$$p(\mathbf{x}) = \sum_{y \in \mathcal{Y}_\mathbf{x}} p(\mathbf{x}, y) \tag{4.5}$$

$$= \sum_{y \in \mathcal{Y}_\mathbf{x}} p_\theta(y) \qquad \text{via Equation 4.4} \tag{4.6}$$

Under this formulation, one natural application of PCFGs is language modeling, or assigning scores (in this case, probabilities) to input sentences (Jurafsky et al., 1995). For most NLP applications, however, it is not the probability of the string that is of interest but rather the best derivation (or set of derivations) associated with input, since the particular grammar rules in each derivation often contain important details about linguistic structure.

The trick involved with using PCFGs for semantic parsing is that we associate each derivation with a unique `LF`, as shown in Figure 4.6 (on the bottom of each derivation). For example, in the derivation considered above, the *interpretation* of this derivation, which we express as $[\![y]\!]^\mathcal{G}$ (see Section 4.4.1 for more details about

Algorithm 8 CKY Recognition Algorithm

Input: CFG $\mathcal{G}$ in Chomsky Normal Form, input $\mathbf{x} = (x_1, .., x_{|\mathbf{x}|})$, start symbol S
Output: TRUE if $\mathbf{x}$ is accepted, FALSE otherwise

$\mathcal{T} \leftarrow \emptyset$ ▷ Initialize chart data structure
for j from 1 up to $|\ \mathbf{x}\ |$ **do**
  for all terminal rules $A \rightarrow \alpha \in \mathcal{G}_R$ **do** ▷ Search for terminal rule matches
    if rule is $A \rightarrow x_j$ **then**
      $\mathcal{T} \leftarrow \mathcal{T} + [j-1, A, j]$
  for i from $j-2$ down to 0 **do** ▷ Search for binary rule matches
    for k from $i+1$ to $j-1$ **do**
      for all binary rules $A \rightarrow BC \in \mathcal{G}_R$ **do**
        if $[i, B, k]$ and $[k, C, j] \in \mathcal{T}$ **then**
          $\mathcal{T} \leftarrow \mathcal{T} + [i, A, j]$
return $[0, S, |\mathbf{x}|] \in \mathcal{T}$

how this interpretation is computed), is the following `LF`:

$$\mathbf{z} = \texttt{kick(purple10)}$$

In doing this, we can then define a conditional distribution over `LF` outputs $\mathbf{z}$ (or entailments when modeling entailment) given inputs $\mathbf{x}$, as in the following:

$$p_\theta(\mathbf{z} \mid \mathbf{x}) \propto \sum_{y \in \mathcal{Y}_\mathbf{x} \mid [\![y]\!]^\mathcal{G} = \mathbf{z}} p_\theta(y) \tag{4.7}$$

which allows us in effect to use the PCFG as a special kind of constrained translation model (with which we can model weighted relations between natural languages and semantic languages as first discussed in Section 1.2).

One inherent difficulty with PCFGs and the computations described above is that the space of derivations $\mathcal{Y}_\mathbf{x}$ can be exponential over the size of each input $\mathbf{x}$ (this is similar to the issue of computing all alignments in the translation models from Section 2.3.1). Often these issues can be overcome by applying standard dynamic programming techniques (as described in the next section), however not all such techniques can be applied when using our model in the manner described above for semantic parsing, as we discuss next.

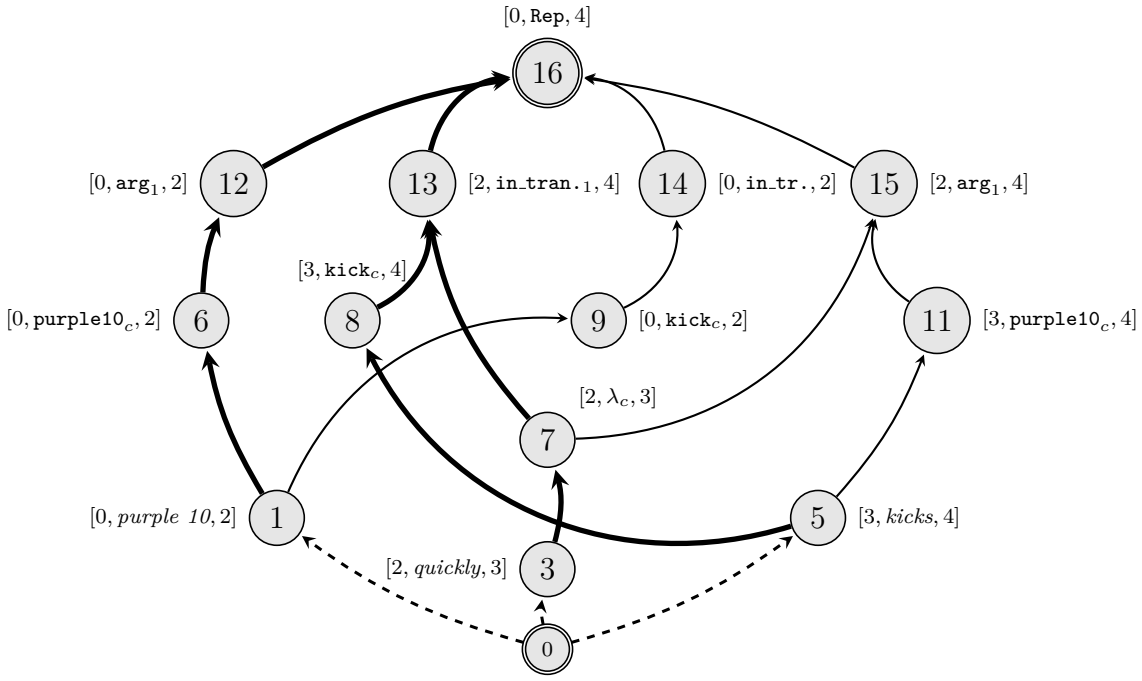


Figure 4.7: An acyclic hypergraph representation of the first two derivations in Figure 4.6, with the shortest path (or tree) shown in bold.

Recognition and Decoding Given a generic PCFG $\mathcal{G}_\theta$ and an input $\mathbf{x}$, the decoding problem involves finding the most probable derivation y^* associated with the input, as expressed below:

$$y^* = \arg\max_{y \in \mathcal{Y}_\mathbf{x}} \Big\{ p(y \mid \mathbf{x}) \Big\} \tag{4.8}$$

which can be reformulated in the following way via the *decoding rule* (Smith, 2011), which is based on the fact that $p(\mathbf{x})$ remains fixed for each candidate y:

$$\begin{aligned}
y^* &= \arg\max_{y \in \mathcal{Y}_\mathbf{x}} \Big\{ p(y \mid \mathbf{x}) \Big\} && \text{Eq. 4.8} \\
&= \arg\max_{y \in \mathcal{Y}_\mathbf{x}} \Big\{ \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} \Big\} && \text{Definition} \\
&= \arg\max_{y \in \mathcal{Y}_\mathbf{x}} \Big\{ p(\mathbf{x}, y) \Big\} && \text{constant } p(\mathbf{x})
\end{aligned}$$

Algorithm 9 Directed Acyclic Hypergraph (DAH) Shortest-Path Search

Input: DAH $\mathcal{H}$, edge labels parameters θ (probabilities, e.g., grammar rule parameters)
Output: Shortest hyperpath

1: $d[V[\mathcal{H}]] \leftarrow \infty$ ▷ Standard initialization (i.e., for DAG SSSP)
2: $\pi[V[\mathcal{H}]] \leftarrow Nil$
3: $d[0] \leftarrow 0$
4: **for** each node $v \in V[\mathcal{H}]$ in sorted order **do**
5: **for** each hyperedge $e = \left(\{u_1, u_2, ..., u_{|e|}\}, v, l\right) \in \texttt{BS}(v)$ **do**
6: $\texttt{score} \leftarrow -\log(\theta_l) + \sum_i^{|e|} d[u_i]$ ▷ hyperedge score computation via θ
7: **if** $d[v] > \texttt{score}$ **then** ▷ Standard relaxation step
8: $d[v] \leftarrow \texttt{score}$
9: $\pi[v] \leftarrow e$
10: **return** FindPath$(\pi, \mid V \mid, 0)$ ▷ Backtrace to find shortest hyperpath

Again, the difficulty here involves efficiently computing all the derivations in $\mathcal{Y}_{\mathbf{x}}$. One way do this is to use a variant of the CKY algorithm shown in Algorithm 8 (Kasami, 1965; Nederhof and Satta, 2010). As presented, the CKY solves the more fundamental problem of *recognition*, or determining if an input string $\mathbf{x}$ is in the language defined by a CFG, which similarly involves searching through all possible derivations. This is done efficiently by using a chart data structure $\mathcal{T}$ and dynamic programming to efficiently search and store all applications of rules in intermediate derivations (lines 4 and 9, see Manning and Schütze (1999) for more details).

The chart data structure $\mathcal{T}$ that results from the CKY search can be interpreted as a special type of directed graph called a directed hypergraph (Gallo et al., 1993), which extends ordinary directed graphs by allowing edges to connect to multiple nodes. In the parsing case, nodes are associated with particular rule applications (i.e., each $[i, R, j]$ from lines 5 and 10 in Algorithm 8) and edges are associated with production rules, as shown in Figure 4.7. With this graph, one can then do decoding by extending the shortest path algorithms for directed graphs (Section 3.4) to hypergraphs (Knuth, 1977; Klein and Manning, 2004; Huang, 2008).

Formally, a directed hypergraph $\mathcal{H} = (V, E)$ consists of a set of nodes V and directed hyperedges E, where each hyperedge e takes the following form (see Huang

(2008) for a more general overview and notation):

$$e = \Big(\{u_1, u_2, ..., u_{|e|}\}, v, l\Big)$$

and consists of a set of *tail nodes* $t(e) = \{u_1, .., u_{|e|}\} \subseteq V$, a *head node* $h(e) = v \in V$ and (for convenience) a label l. In the case of the CKY algorithm, the hypergraph that is generated is an acyclic directed hypergraph, which has the property (as with DAGs) that nodes can be sorted into numerical (topological) order. The associated shortest path algorithm, therefore, is nearly identical to the one for DAGs (see Algorithm 4), and is shown in Algorithm 9 (where $\texttt{BS}(v) = \{e \in E \mid h(e) = v\}$[1] and in the parsing case, labels are used to identify grammar rules R associated with each e). The shortest path (or best derivation tree) can then be constructed by moving backwards (via the FindPath routine) from the final node (or the start node S) to the source node using the *predecessor* π.

Returning to the use of our PCFG as a semantic parser, the decoding problem (i.e., finding the best `LF` or entailment $\mathbf{z}^*$ given $\mathbf{x}$) can be described in the following

$$\mathbf{z}^* = \arg\max_{\mathbf{z}} \Big\{p_\theta(\mathbf{z} \mid \mathbf{x})\Big\} \tag{4.9}$$

and is at first glance more difficult than Equation 4.8 given that computing this requires finding all *valid derivations* $\{y \mid \mathbf{z} = [\![y]\!]^{\mathcal{G}}\}$ as per Equation 4.7. The problem is that computing each valid derivation often requires a non-local combination of rules in the target tree, which cannot be accomplished using dynamic programming. For example, computing the `LF kick(purple10)` from the first derivation tree in Figure 4.6 requires combining information from the two subtrees rooted by $\texttt{purple10}_c$ and $\texttt{kick}_c$, which are not adjacent in $\mathcal{T}$. Therefore, computing valid trees requires enumerating an intractable number of trees and interpreting them.

One way to get around this is to approximate this search by taking $\mathbf{z}^*$ to be the

[1] The *backwards star* $\texttt{BS}(v)$, which in ordinary directed graphs denotes the set of incoming edges to v, has a *forward* variant $\texttt{FS} = \{e \mid v \in t(e)\}$ that is analogous to `Adj` in Algorithm 4. We note that for DAG SSSP search, either type of traversal order can be used, whereas `BS` traversal is more straightforward for hypergraphs (see discussion in Huang (2008)).

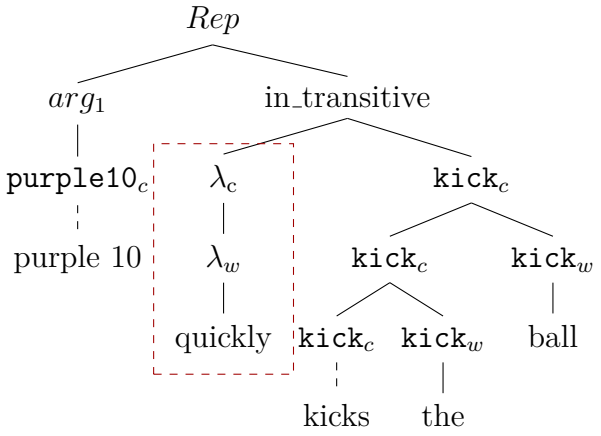


Figure 4.8: An example derivation (simplified) in the base semantic grammar with a gap rule λ_c applied over the modifier *quickly*.

interpretation of the most probable derivation:

$$\mathbf{z}^* \approx [\![y]\!]^{\mathcal{G}} = \arg\max_{y \in \mathcal{Y}_{\mathbf{x}}} \Big\{ p_\theta(y) \Big\} \tag{4.10}$$

which is what we do when evaluating our models. While this works well for decoding at test time, it is still a problem when estimating our models (i.e., finding the expected counts of rules in valid derivations during the training phase). We discuss this more in Section 4.4.3, and propose a simple EM bootstrapping method that similarly involves sampling the best derivations via k-shortest path decoding (using variations of the DAG k-SSSP algorithms used in Section 3.4).

4.4.1 Rule Extraction and $[\![\cdot]\!]^{\mathcal{G}}$

As already discussed, rule extraction is the process of constructing the grammar rules R needed for generating $\mathbf{z}$'s from input $\mathbf{x}$. We start by describing rule construction for grammars that generate LFs (or what we call *base semantic grammars*) and return to how rule extraction works for modeling inference in Section 4.4.2 (the *inference grammars*). In this first case, such rules are constructed automatically using a small set of rule templates defined over the target set of LFs, as done

in Börschinger et al. (2011) (BB). The basic idea in BB is to break down all LFs in the target corpus of the form $\mathtt{R}(x, y)$ into the following production rules:

$$\mathtt{S} \rightarrow \mathtt{R}(\mathtt{x}, \mathtt{y})$$
$$\mathtt{R}(\mathtt{x}, \mathtt{y}) \rightarrow \left\{\mathtt{R}_c \;\; \mathtt{x}_c \;\; \mathtt{y}_c\right\}$$

where the lhs of the second rule is a representation of the full LF, and the rhs consists of all orderings (as indicated by $\{\cdot\}$) of the constituent parts of the LF expressed as grammar symbols (i.e., the relation name R and the arguments, all marked here as $\mathtt{X}_c$). Each non-terminal $\mathtt{X}_c$ is then associated with a word rule $\mathtt{X}_w$, that rewrites to all unigrams in the target corpus via a left-recursive rule that models a unigram Markov-process (Johnson and Goldwater, 2009) (e.g., the rule sequence for *kicks the ball* in Figure 4.8):

$$X_c \rightarrow X_w$$
$$X_c \rightarrow X_c X_w$$
$$X_w \rightarrow w \mid w \in \text{Corpus}$$

Using this basic idea, additional structure and information can be added into the grammar as needed. For example, BB use word order rules that make explicit the different orderings of constituent rules in $\{\cdot\}$, as well as more complex word rules that allow for modeling empty (or skip) words λ_w. We adopt both of these ideas, and pad all nodes $\mathtt{X}_c$ with a *gap rule* λ_c that allows the model to learn larger spans of unanalyzed text. For example, in Figure 4.8 the model learned that *quickly* is not analyzed in the target LF, which is information that can be used by our inference model (described in Section 4.4.2) to reason about the semantics of these gaps. Rather than representing full LFs in the grammar as atomic symbols, we also assign more abstract role types to the concepts X_c (e.g., $\text{arg}_1, \text{in\_transitive}$ in Figure 4.8), to make the rules more generalizable (see Appendix C for a full description of the rule templates we use in our experiments).

As discussed above, each derivation tree y can be interpreted to a unique LF via

an interpretation function $[\![\cdot]\!]^{\mathcal{G}}$:

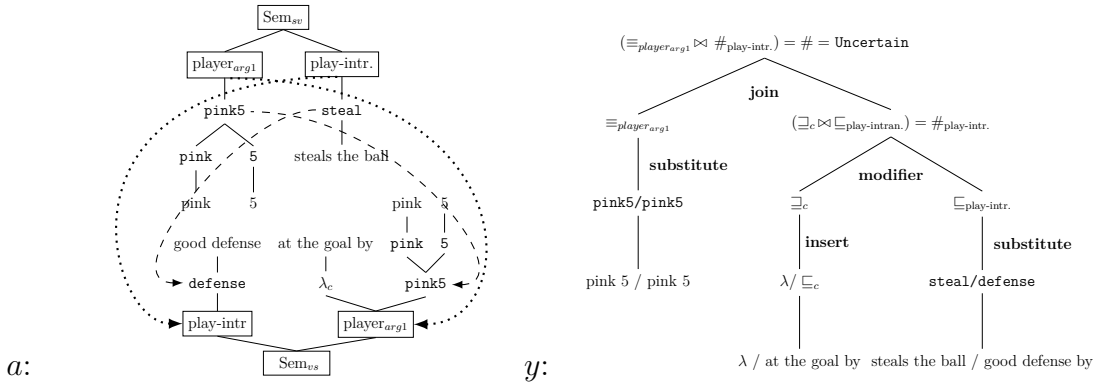
$$[\![\cdot]\!]^{\mathcal{G}} : y \ (\textit{derivation}) \rightarrow \mathbf{z} \tag{4.11}$$

In our case, this function works by deterministically mapping each grammar symbol $\texttt{X}_c$ to an atomic logical symbol, and combining these symbols in a way that is consistent with the assigned roles. For example, $\texttt{kick}_c$ in Figure 4.8 is mapped to the symbol `kick` and assigned to the main predicate slot given the *in_transitive* role, and $\texttt{purple10}_c$ is mapped to `purple10` as assigned as the first argument slot given the arg_1, which results in the `LF` representation `kick(purple10)`.

An important feature of the resulting grammars is that they overgenerate (as shown in Figure 4.8); given a text input, the grammar will generate a large space of possible derivations, many of which interpret to incorrect `LF`s. By assigning weights to these rules and formalizing the model as a PCFG, the learning problem reduces to a grammatical inference problem, or finding a grammar $\mathcal{G}_\theta$ with parameters θ that is able to distinguish correct derivations (i.e., derivations that have the correct interpretations) from incorrect derivations. Under a hypergraph approach, we can equivalently describe the learning problem as finding a model that is able to identify the correct paths through graphs such as the one in Figure 4.7 (see Section 4.4.3 for more details about learning).

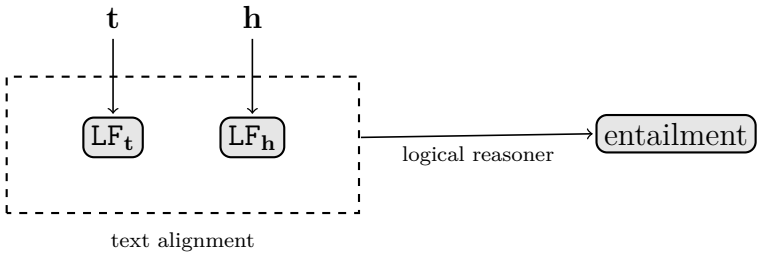
4.4.2 Natural Logic and Inference Grammars

Given the *base semantic grammars* described in the previous section, we can build a semantic parser that (standardly) translates text to output `LF`s, however the resulting derivation trees still have gaps (i.e., unanalyzed spans of text as shown in Figure 4.8) and our model continues to lack the background knowledge needed for reasoning about entailment. As already proposed, we aim to learn this missing information by extending our training corpus to include pairs $\mathbf{x} = (\mathbf{t}, \mathbf{h})$ annotated with entailment information. With this information, our approach works in the following way: align the related spans of text in $\mathbf{t}$ and $\mathbf{h}$ and apply logical reasoning over these spans to construct a proof of an entailment, as sketched below:



$$\Big((\mathbf{t} = \textit{pink 5 steals the ball}, \mathbf{h} = \textit{good defense at the goal by pink 5}), \mathbf{z} = \texttt{Uncertain} \Big)$$

Figure 4.9: An end-to-end example produced by our inference grammar model.



This idea is further illustrated in Figure 4.9, where an alignment a between $\mathbf{t}$ and $\mathbf{h}$ is computed by heuristically matching related roles in the semantic parse for each sentence (generated using the base semantic grammars described above). The associated spans of aligned text are then provided to a logical reasoner that remaps the aligned spans to logical symbols, then generates a structured proof y over these symbols that corresponds to a unique entailment judgement. Importantly, gap rules in $\mathbf{t}$ or $\mathbf{h}$ (e.g. *at the goal by* in $\mathbf{h}$, marked as λ_c) and unaligned arguments are matched to the empty string λ so that the logical model can reason about the semantics of *inserting* or *deleting* expressions in $\mathbf{h}$ and $\mathbf{t}$.

Relation	Symbol	Set Definition	First-order Logic	RTE label
forward entail	$R \sqsubseteq S$	$R \subset R$	$\forall x.\big[R(x) \rightarrow S(x)\big]$	`entail`
reverse entail	$R \sqsupseteq S$	$R \supset S$	$\forall x.\big[S(x) \rightarrow R(x)\big]$	`uncertain`
equivalence	$R \equiv S$	$R = S$	$\forall x.\big[R(x) \leftrightarrow S(x)\big]$	`entail`
alternation (negation)	$R \mid S$	$R \cap S = \emptyset \wedge R \cup S \neq D$	$\forall x.\neg\big[R(x) \wedge S(x)\big]$	`contradict`
independence	$R \# S$	(all other cases)	–	`uncertain`

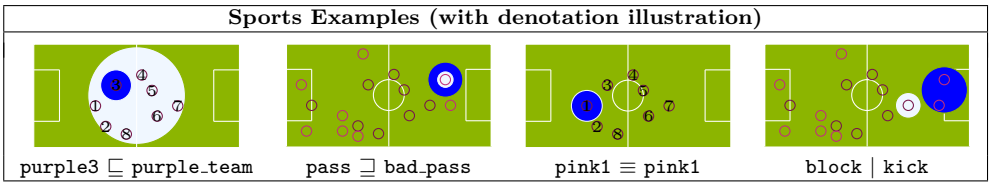


Figure 4.10: A description of relations used from the natural logic calculus (top) with examples (bottom) from Sportscaster.

Natural Logic Calculus

In order to do modeling of this kind, we need a logical calculus that can perform reasoning over spans of text. Since our main goal is to learn proofs and the background knowledge that drives the proofs, such a model should also support uncertainty. Given these constraints, we use a fragment of the natural logic calculus defined in MacCartney and Manning (2009, 2008). In our simplified version, the model has two components: 1) a set of relations that define abstract semantic relationships between concepts (i.e., primitive symbols in our target `LF` representations) and 2) a set of inference rules that can compose relations. The full set of semantic relations are shown in Figure 4.10, along with a definition of their meaning in set theory and first-order logic (or FOL, following Pavlick et al. (2015)). For example, the following relation between `pass` and `kick`:

$$\texttt{pass} \sqsubseteq \texttt{kick}$$

can be read in FOL as a general implication (van Benthem, 1986): for all events e involving passing, e also involves kicking. Assuming that we are given two pairs of relations, `pink5` $\sqsubseteq$ `pink_team` and `pass` $\sqsubseteq$ `kick`, the join rule $\bowtie$ defined in

Figure 4.11 then composes these two relations to derive a new relation:

$$\Big(\texttt{pink5} \sqsubseteq \texttt{pink\_team}\Big) \bowtie \Big(\texttt{pass} \sqsubseteq \texttt{kick}\Big) = \sqsubseteq$$

which in this case allows us to conclude that *pink5 passed* (forward) entails that *the pink team kicked* (on the assumption that `pink5` forward entails `pink_team` and `pass` forward entails `kick`). Joins can be applied an arbitrary number of times; for example, we might continue by adding $\lambda \sqsupseteq \texttt{quickly}$ to model in the RTE context the insertion of a modifier *quickly* on the **h** side:

$$\Big(\texttt{pink5 pass} \sqsubseteq \texttt{pink\_team kick}\Big) \bowtie \Big(\lambda \sqsupseteq \texttt{quickly}\Big) = \#$$

which results in a new relation #. This process can continue further until all target relations have been consumed, which will result in a final semantic relation and entailment (see Figure 4.10 to see the mapping from relations to RTE labels).

We note that our simplified model uses only a subset of the seven relations from MacCartney and Manning (2009), since these additional relations were not needed to model the types of inferences we encountered in the Sportscaster domain. As a cautionary note, we also point out that our model fails to capture various complex inferences. For example, assuming $\texttt{every} \equiv \texttt{every}$ and $\texttt{company} \sqsupseteq \texttt{small\_company}$, our model cannot generate the following entailment using the join inference rule:

$$\texttt{every company} \sqsubseteq \texttt{every small\_company}$$

since this particular inference is related to special properties of `every`, which convert $\sqsupseteq$ inferences to $\sqsubseteq$ when doing composition in this context. To handle this, the full natural logic calculus has a *projectivity* mechanism that defines how certain constructions alter the inferences of arguments in such contexts. In our simple mode and domain, *projectivity* is limited to a single rule that always projects negations | up the proof tree in order to capture the following inferences:

$$\texttt{pink5 kick} \mid \texttt{purple\_team pass}$$

where under the assumption that $\texttt{pink5} \mid \texttt{purple\_team}$ and $\texttt{kick} \sqsupseteq \texttt{pass}$, our

$\bowtie$	$\equiv$	$\sqsubseteq$	$\sqsupseteq$	$\mid$	#
$\equiv$	$\equiv$	$\sqsubseteq$	$\sqsupseteq$	$\mid$	#
$\sqsubseteq$	$\sqsubseteq$	$\sqsubseteq$	#	$\mid$	#
$\sqsupseteq$	$\sqsupseteq$	#	$\sqsupseteq$	#	#
$\mid$	$\mid$	#	$\mid$	#	#
#	#	#	#	#	#

Figure 4.11: The join inference rule $\bowtie$ table for the set of relations in Figure 4.10.

join rule would incorrectly assign # (or an `uncertain` entailment). This is again motivated by the types of predicates we model in the Sportscaster domain, which all tend to have the projectivity properties of *functional relations* (Russell (1995), see MacCartney (2009)[Chapter 6.2.5] for more discussion).

Inference Grammars and Alignment

Given the tree-like nature of the natural logic proofs described above, our idea is to represent the inference steps as CFG rewrite rules, as shown in Figure 4.12[2]. Under this approach, relations between pairs of concepts are rules where the rhs contains the pair of ordered concepts (delimited by `/`) and the lhs contains their resulting relation. The same idea applies to our inference rule $\bowtie$: the rhs consists of two relations and the lhs contains the result of joining these relations. As before, additional structure can be added to the grammar as needed, such as information about the types of concepts being compared and composed (see Appendix C for a complete list of the rules we use, as well as Figure 4.15).

In particular, we use an additional set of *gap rules*, as shown in Figure 4.12, that model whether certain types of insertions/deletions are forward-entailing (represented using $\sqsubseteq_c$) or non-entailing ($\equiv_c$). In the first case, this includes adverbial modifiers such as *quickly* in *quickly kicked* which modify entailment, whereas *the*

[2]This particular grammar formulation can be regarded as a type of inversion transduction grammar (Wu, 1997), or a *simple transduction grammar* (Lewis II and Stearns, 1968), where each terminal rule is marked with an input and output symbol and non-terminals are the same as in ordinary CFGs. Recognition and decoding with these models is equivalent to ordinary CFG parsing as described above (Melamed, 2004; Lopez, 2008).

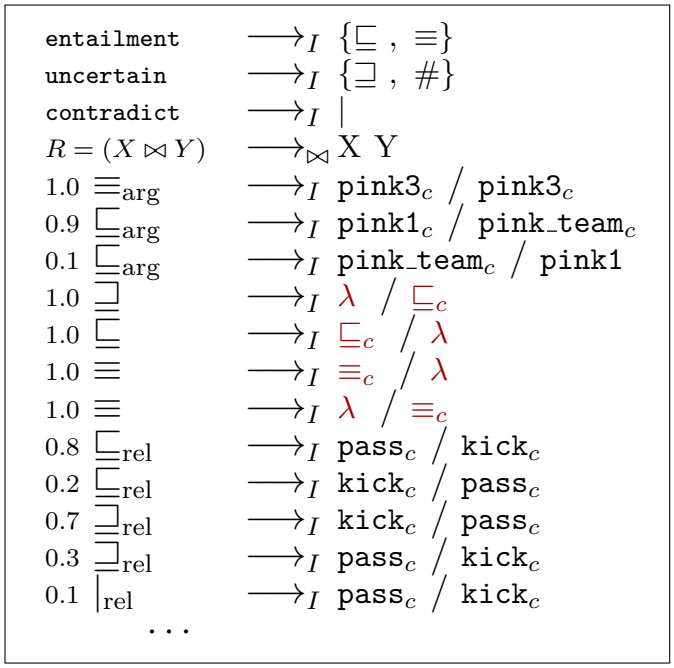


Figure 4.12: An example inference grammar for the Sportscaster domain with gap rules shown in red.

ball in *kick the ball* does not appear to effect entailment. To model the subtle differences between different concept senses, we also mark symbols with latent sense labels. For example, in the following rule:

$$\sqsubseteq_{rel} \rightarrow \texttt{kick}_{c1} \ / \ \texttt{kick}_{c2}$$

we have two senses for `kick`, which we can use to model entailments between *kick the ball* and *score a goal* (which are both annotated as `kick` in Sportscaster).

As discussed at the onset, given a pair $\mathbf{x} = (\mathbf{t}, \mathbf{h})$, we can generate trees in this grammar by heuristically aligning related spans in $\mathbf{t}$ and $\mathbf{h}$ (see again Figure 4.9, and Appendix C for more details), then by labeling each part of the the spans with concept labels and applying the grammar rules. The interpretation of a given derivation (proof tree) $[\![y]\!]^{\mathcal{G}}$ is the entailment provided at the top node of each tree. As shown in the grammar in Figure 4.12, the concept labels $\texttt{X}_c$ are the same

as in our LF semantic parser, which allows us to create a single grammar by combining the inference rules with the base semantic grammar (and therefore use the same learned concept mapping rules as in our main semantic grammar). In our experiments, the decision to jointly train the sem and infer models using a single grammar is based on the following modeling assumption:

- **Joint entailment modeling**: When learning a semantic parser, improvements on learning the correct entailments should help improve (and are tied to) learning translations to LFs, and vice versa.

As with the base semantic grammars, an important feature of the inference grammars described above is that they overgenerate; given an input, the grammar will generate a large space of possible proofs, many of which interpret to the wrong entailment. This is largely due to the fact that we do not know the correct relations between the underlying concepts and modifiers and start by assuming all possibilities. For example, since we do not know the relation between pass and kick, we start with the following three rules:

$$\sqsubseteq_{rel} \rightarrow \texttt{pass}_c \ \texttt{/} \ \texttt{kick}_c$$
$$\sqsupseteq_{rel} \rightarrow \texttt{pass}_c \ \texttt{/} \ \texttt{kick}_c$$
$$|_{rel} \rightarrow \texttt{pass}_c \ \texttt{/} \ \texttt{kick}_c$$

The key idea is that by interpreting these grammars as PCFGs, we can then associate weights with individual rules of this type and learn the correct relations by training our grammar on example entailments. In this case, the goal is to learn that the first rule should have a higher weight than the other two rules since pass forward entails kick as inferred from its appearance in example proofs. Given that particular orderings of join inferences can effect the resulting entailments (MacCartney, 2009), the PCFG approach also allows for learning optimal inference combinations.

Under the PCFG formulation, our model can therefore handle probabilistic inference, which distinguishes it from most other formulations of natural logic (for a similar idea, see Angeli and Manning (2014)). While our particular rule templates might seem arbitrary at first glance, we note that the probabilistic logic that re-

sults from this formulation seems to have a sensible semantics. In addition, our use of grammar representations allows us to apply efficient search strategies from parsing to the problem of entailment search. For example, the probability of an `entail` using Equation 4.7 is given by the following:

$$p_\theta(\mathbf{z} = \texttt{entail} \mid \mathbf{x} = (\mathbf{t}, \mathbf{h})) = \sum_{y \in \mathcal{Y}_\mathbf{x} \mid [\![y]\!]^\mathcal{G} = \texttt{entail}} p(y \mid \mathbf{x})$$

and is interpreted as all proofs between $\mathbf{t}$ and $\mathbf{h}$ that evaluate to *entailment* (within the space of all possible proofs and across all individual semantic interpretations of $\mathbf{t}$ and $\mathbf{h}$). Since our approach involves a heuristic alignment between $\mathbf{t}$ and $\mathbf{h}$ (and hence is not burdened by having to search all possible alignments), the basic proof search is therefore bound to the complexity of ordinary recognition (e.g., using the CKY algorithm), or $O(|\mathbf{x}|^3 \cdot |\mathcal{G}_R|)$. Since the interpretation in these grammars only requires reading a single node, computing the above equation can be done exactly with the same complexity using the inside algorithm (Lari and Young, 1990).

4.4.3 Learning

As discussed in the previous section, we model `sem` and `infer` using a joint PCFG model that uses the rules described above. To learn this model, we perform maximum likelihood estimation (MLE) over our parallel dataset $D = \{(\mathbf{x}^{(d)}, \mathbf{Z}^{(d)})\}_{d=1}^{|D|}$, consisting both of parallel semantic parsing data and parallel inference data. Formally, the objective is to find grammar parameters θ^* that maximize the following (where we use $\mathcal{C}_{(d)}$ to denote the set of *valid (interpretable) derivations* relative to each training annotation $\mathbf{Z}^{(d)}$ and input $\mathbf{x}^{(d)}$: $\{y \mid y \in \mathcal{Y}_{\mathbf{x}^{(d)}} \wedge [\![y]\!]^\mathcal{G} \in \mathbf{Z}^{(d)}\}$):

$$\theta^* = \max_\theta \log \prod_{d=1}^{|D|} \left[p_\theta(\mathbf{z}^{(d)} \mid \mathbf{x}^{(d)}) \right] \tag{4.12}$$

$$= \max_\theta \sum_{d=1}^{|D|} \log \left[\sum_{y \in \mathcal{C}_{(d)}} p_\theta(y) \right] \qquad \text{via Eq. 4.7} \tag{4.13}$$

To optimize this objective, we use a variant of the EM algorithm (for a review of EM, refer back to Section 2.3.1). As in normal EM for PCFGs (Lari and Young,

Algorithm 10 EM Grammar Boostrapping

Input: Grammar $\mathcal{G}$ with parameters θ, dataset D, interp. function $[\![\,]\!]^{\mathcal{G}}$, KBEST function with k
Output: Learned parameters θ

$\theta^0 \leftarrow$uniform initialization
$t \leftarrow 0$
repeat
  $c(N \to \beta) \leftarrow 0, \forall N \to \beta \in \mathcal{G}_R$ ▷ Initialize counters to collect rule counts
  $b(N) \leftarrow 0, \forall N \to \beta \in \mathcal{G}_R$
  for $(\mathbf{x}^{(d)}, \mathbf{Z}^{(d)})$ from $d = 1$ up to $\mid D \mid$ **do** ▷ E-Step: evaluate $p_\theta(\mathbf{Z} \mid \mathbf{x}) \sim$ KBEST
    $v \leftarrow [\,], n \leftarrow 0$
    for $(y, p) \in \text{KBEST}_{(d)}(\mathbf{x}^{(d)}, \mathcal{G}, \theta^t, k)$ **do** ▷ Find candidate derivation y, $p = p_\theta(y \mid \mathbf{x})$
      if $[\![y]\!]^{\mathcal{G}} \in \mathbf{Z}^{(d)}$ **then**
        $n \leftarrow n + p$ ▷ Add valid derivations with scores
        $v \leftarrow v + (y, p)$
    for $(y, p) \in v$ **do** ▷ Count rules in valid derivations
      for $N_i \to \beta_i$ from $i = 1$ up to $|y|$ **do**
        $c(N_i \to \beta_i) \leftarrow c(N_i \to \beta_i) + \frac{p}{n}$ ▷ Normalize using n to create prob. distr.
        $b(N_i) \leftarrow b(N_i) + \frac{p}{n}$
  for $N \to \beta \in \mathcal{G}_R$ **do** ▷ M-step: perform MLE updates
    $\theta^{t+1}_{N \to \beta} \leftarrow \frac{c(N \to \beta)}{b(N)}$
  $t \leftarrow t + 1$
until converged
return θ^t

1990; Lafferty, 2000), the E-step involves finding the expected counts of individual production rules R in all latent derivations (or in our case, all valid derivations $\mathcal{C}_{(d)}$) given D and some posterior distribution $p_{\theta^t}(y \mid \mathbf{x}^{(j)})$:

$$c(R; D) = \sum_{d=1}^{|D|} \left[\sum_{y \in \mathcal{C}_{(d)}} p_{\theta^t}(y \mid \mathbf{x}^{(d)}) \sum_{i}^{|y|} \delta(r_i, R) \right] \tag{4.14}$$

Using these counts, the M-Step then involves performing ordinary MLE updates, and the process then repeats until a convergence point:

$$\theta^{t+1}_{N \to \beta} = \frac{c(N \to \beta; D)}{\sum_{\beta'} c(N \to \beta'; D)} \tag{4.15}$$

As before, the main problem involves efficiently computing the set of valid derivations $\mathcal{C}_{(d)}$, since our interpretation function $[\![\cdot]\!]^{\mathcal{G}}$ involves non-local combinations

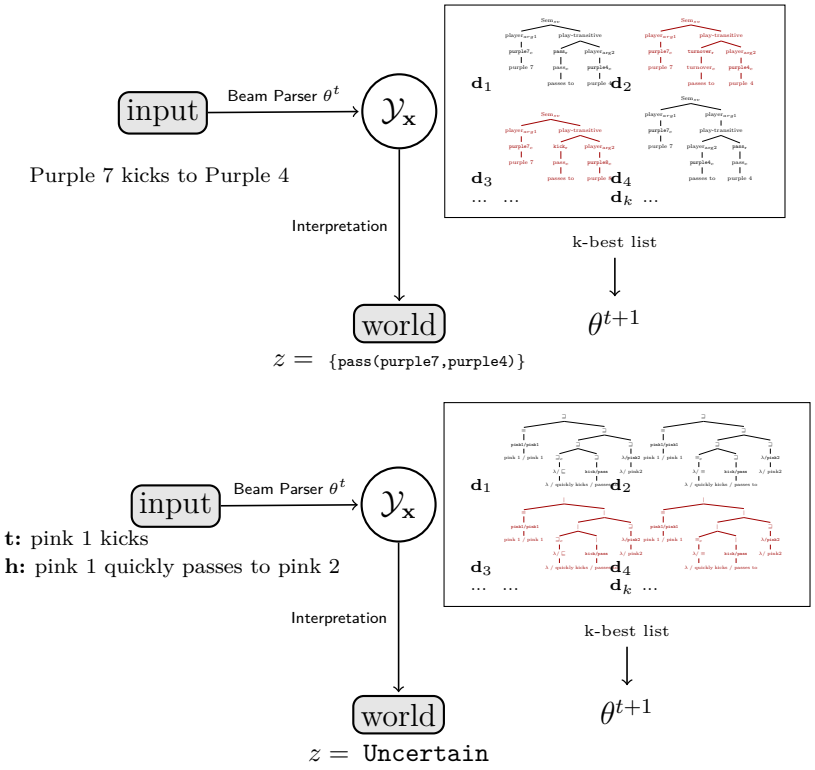


Figure 4.13: An illustration of EM bootstrapping for semantic parsing and learning from entailment, where green shows the valid derivations.

of derivation rules. We get around this by approximating each $\mathcal{Y}_\mathbf{x}$ in $\mathcal{C}_{(d)}$ with a k-best list of derivations $\text{KBEST}_{(d)} \approx \mathcal{Y}_\mathbf{x}$ (Angeli et al., 2012). Under the hypergraph approach outlined in Section 4.4, sampling the k-best derivations can be achieved by extending the branching $k-$SSSP method used in Algorithm 7 and Section 3.4.3 for DAGs to hypergraphs, as done in Nielsen et al. (2005). Given special features of parsing, however, more efficient methods based on hypergraphs have been developed, notably the lazy k-best algorithm from Huang and Chiang (2005), which is what we use in our experiments.

The full training algorithm is shown in Algorithm 10, along with an illustration in Figure 4.13 of detecting the valid derivations (line 9) and computing new parameters θ^{t+1} based on collected rule counts (lines 16-17). As discussed in Liang et al.

(2011), the idea is that learning starts in an unguided manner and improves over time by *bootstrapping* off of the easy examples. When training with the entailment pairs, the distinctions being made for modeling inference (e.g. sense distinctions, modifier types) and the word rules being used inform and reinforce the learning of the base semantic grammar. As the quality of the semantic parser improves, so should the quality of the background knowledge (i.e. semantic relations) used to generate the natural logic proofs.

4.5 Experimental Setup

In this section, we provide more details about the Sportscaster dataset and a new Sportscaster inference corpus that we created for modeling and evaluating entailment. We also describe our main experimental setup, which consists of two tasks: 1) the standard Sportscaster semantic parsing task, and 2) a new RTE-style entailment recognition task. We end the section by providing additional implementation and model details (for more information, see also Appendix C).

4.5.1 Datasets

Sportscaster The Sportscaster corpus (Chen and Mooney, 2008a) consists of 4 simulated Robocup soccer games annotated with human commentary. The English portion includes 1,872 sentences paired with sets **Z** of logical meaning representations. On average, each training instance is paired with 2.3 meaning representations. The representations have 46 different types of concepts, consisting of 22 entity types and 24 event (and event-like) predicate types (see Figure 3.9 for a description and implementation of the Sportscaster language).

While the domain has a relatively small set of concepts and limited scope, reasoning in this domain still requires a large set of semantic relations and background knowledge. From this small set of concepts, the inference grammar described in Section 4.4.2 encodes around 3,000 inference rules. Since soccer is a topic that most people are familiar with, it is also easy to get non-experts to provide judgements about entailment.

Task 1: Semantic Parsing	Match F_1(%)
LexDecoder (Chapter 3)	40.3
Kim and Mooney (2010)	74.2
Chen et al. (2010)	80.1
Best Seq2Seq model (Chapter 3)	83.4
Börschinger et al. (2011)	86.0
Gaspers and Cimiano (2014)	88.7
base semantic grammar (BSG) only	**95.7**
BSG + inference grammar (IG)	95.8
BSG + IG + More Data	**96.3**

Task 2: Inference Task	Accuracy (%)
Majority Baseline	33.1
RTE classifier	52.4
Naïve Inference	59.6
SVM Flat Classifier	64.3
inference grammar (Lex. Inference Only)	72.0
inference grammar (Full)	**73.4**
inference grammar + More Data	72.3

Table 4.1: Results on the semantic parsing (top) and inference (bottom) cross validation experiments (averaged over all folds)

Extended Inference Corpus The extended corpus consists of 461 unaligned pairs of texts from the original Sportscaster corpus annotated with sentence-level entailment judgements (as first shown in Figure 4.2). We annotated 356 pairs using local human judges an average of 2.5 times using a version of the elicitation instructions for RTE from Snow et al. (2008). Following (Dagan et al., 2005), we discarded pairs without a majority agreement, which resulted in 306 pairs (or 85% of the initial set). We also annotated an additional 155 pairs using Amazon Mechanical Turk, which were mitigated by a local annotator.

In addition to this core set of 461 entailment pairs, we separately experimented with adding unlabeled data (i.e., pairs without inference judgements) and ambiguously labelled data (i.e., pairs with multiple inference judgements) to train our inference grammars (shown in the results as *More Data* in Table 4.1) and test the flexibility of our model. This included 250 unlabeled pairs taken from the original dataset, as well as 592 (ambiguous) pairs created by deriving new conclusions from the annotated set. This last group was constructed by exploiting the transitive nature of various inference relations and mapping pairs with matching labels in training to {`Entail`,`Unknown`}.

4.5.2 Evaluation

We perform two types of experiments: first, a semantic parsing experiment (Task 1 in Table 4.1) to test our approach on the original task of generating Sportscaster `LF` representations. In addition, we introduce a new inference experiment (Task 2) to test our approach on the problem of detecting entailments between unobserved sentence pairs using our inference grammars.

For the semantic parsing experiment, we follow exactly the setup of Chen and Mooney (2008a): 4-fold cross validation is employed by training on all variations of 3 games and evaluating on a left out game. Each representation produced in the evaluation phrase is considered correct if it matches exactly a gold representation and (standardly) F_1 score is reported[3].

The second experiment imitates an RTE-style evaluation and tests the quality of the background knowledge being learned using our inference grammars. Like in the semantic parsing task, we perform cross-validation on the games using both the original data and sentence pairs to jointly train our models, and evaluate on left-out sets of inference pairs. Each proof generated in the evaluation phrase is considered correct if the resulting inference label matches a gold inference. We report on the accuracy of predicting the correct entailment label (within the set { `entail`, `contradict`, `unknown/compatible`}).

4.5.3 Implementation and Model Details

As already discussed, we implemented the learning algorithm shown in Algorithm 10 using the k-best algorithm of Huang and Chiang (2005) (i.e., for the KBest computation in line 8) with a uniform beam size k of 1,000. Following Angeli et al. (2012), we also smoothed rule counts (line 10) by using an additive prior α set to 0.05 for lexical word rules and 0.3 for non-lexical rules. To get good initial estimates of word and concept mapping rules, we pre-trained the joint `LF` and inference grammars by first training the base semantic grammars on the original semantic parsing data for 3 iterations. Lexical rule probabilities were also initialized using co-occurrence statistics estimated using an IBM Model1 word aligner

[3] As with Börschinger et al. (2011), since our grammar model parses every sentence, precision and recall are identical, making F_1 identical to accuracy.

(uniform initialization otherwise).

In the inference grammars, 5 additional senses were added to the most frequent event predicates. In terms of other added background knowledge, we made the default assumption that player terms (i.e., `purple1`,`pink1`,...) have a negation relation | with other player terms that they do not match, and assumed all possible semantic relations between all other types (see Appendix C for more details).

4.6 Experimental Results and Discussion

In this section, we detail the main results featured in Table 4.1 for both tasks, and provide some qualitative analysis on the resulting models.

Task 1: Semantic Parsing

We compare the results of our base semantic parser model with previously published semantic parsing results (including some of the experiments from Section 3.5). While our grammar model simplifies how some of the knowledge is represented in grammar derivations (e.g., in comparison to Börschinger et al. (2011)), the set of output representations or interpretations is restricted to the original Sportscaster formal representations making our results fully comparable. As shown, our base grammar (shown as *base semantic grammar* (only) in Table 4.1) strongly outperforms all previously published results even without the additional inference data and rules. Since our approach is similar to Börschinger et al. (2011), one takeaway is that better rule extraction seems to go a long way in improving accuracy, and might help to improve models such as the Seq2Seq model from the last chapter.

We also show the performance of our inference grammars on the semantic parsing task after being trained with additional inference sentence pairs. This was done under two conditions: when the inference grammar was trained using fully labeled inference data and unlabeled/ambiguously labeled data (*more data*). While not fully comparable to previous results, both cases achieve nearly the same results as the base grammar, indicating that our additional training setup does not lead to an improvement on the original task (but nonetheless has minimal effect of the resulting accuracy).

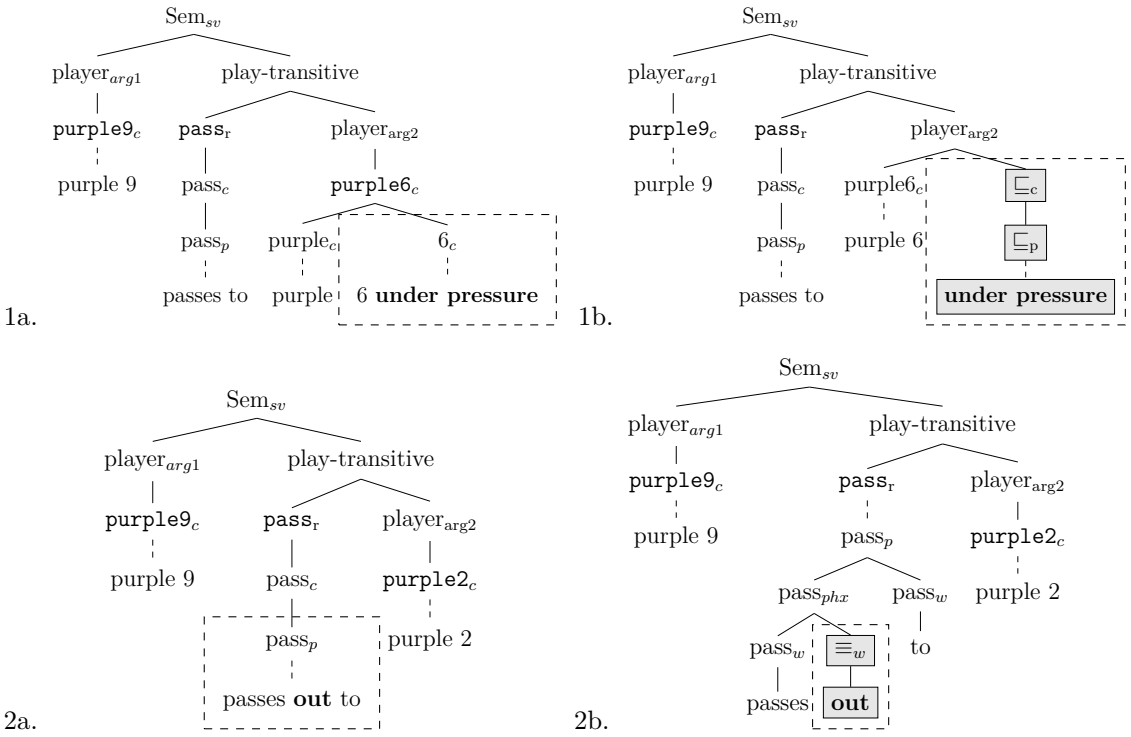


Figure 4.14: Example parse trees (1,2) before (a) and after (b) training on the extended inference corpus (new inferences shown in gray boxes).

Task 2: Inference Task

The main result of this chapter is the performance of our inference grammars on the inference task. For comparison, we developed several baselines, including a *Majority Baseline* (i.e., guess the most frequent inference label from training). We also use an *RTE (max-entropy) classifier* that is trained on the raw text inference pairs to make predictions. This classifier uses a standard set of RTE features (e.g., word overlap, word entity co-occurrence/mismatch). Both of these approaches are strongly outperformed by our main inference grammar (or *inference grammar (Full)*).

The *Naïve Inference* baseline compares the full Sportscaster representations generated by our semantic parser for each sentence in a pair and assigns an `entail` for representations that match and a `contradict` otherwise (as first discussed in

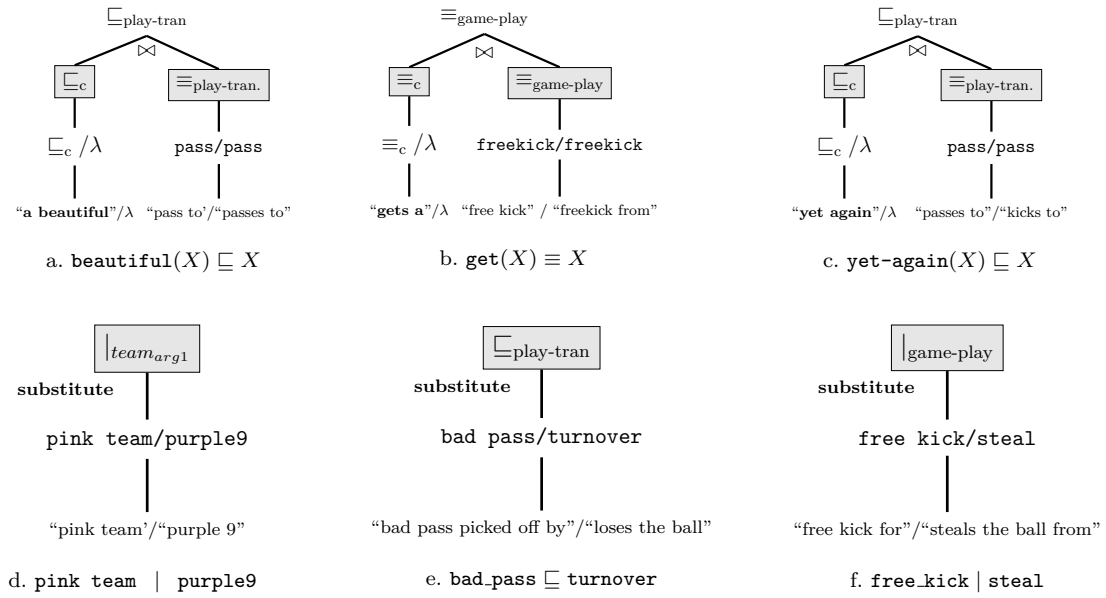


Figure 4.15: Example proof trees involving construction-based (top) and lexical-based (bottom) inferences generated by our model.

Section 4.3.2). This baseline compares the inferential power of the original representations (without background knowledge and more precise labels) to the inferential power of the inference grammars. The strong increase in performance suggests that important distinctions that are not captured in the original representations are indeed being captured in the inference grammars.

We tested another classification approach using a *Flat Classifier*, which is a multi-class SVM classifier (Joachims, 2002) that makes predictions using features from the input and part of the inference model. Such input includes both sentences in a pair, their parse trees and predicted semantic labels, and the alignment between the sentences. In Figure 4.9, for example, this includes all of the information excluding the proof tree in y. This baseline aims to test the effect of using hierarchical, natural logic inference rules as opposed to a *flat* or linear representation of the input, and to see whether our model learns more than the just the presence of important words that are not modeled in the original representations. Features include the particular words/phrases aligned or inserted/deleted, the category of

sense/context error:

1. **t**: *pink9 shoots*
   h: *pink9 shoots for the goal*
   z: `entail` (model prediction: `uncertain`)

semantic parser and alignment error:

2. **t**: *purple8 steals the ball back*
   h: *purple8 steals the ball from pink6*
   z: `uncertain` (model prediction: `contradict`)

alignment and modifier error:

3. **t**: *A goal for the purple team*
   h: *And the purple team scored another goal*
   z: `uncertain` (model prediction: `entail`)

Figure 4.16: Example cases where our inference grammars fail.

these words/phrases in the parse trees, the rules in both parse trees and between the trees, the types of predicates/arguments in the predicted representations and various combinations of these features. This is also strongly outperformed by our main model, suggesting that the natural logic system is learning more general inference patterns.

Finally, we also experimented with removing insertions and deletions of modifiers from alignment inputs to test the effect of only using lexical knowledge to solve the entailment problems (*Lexical Inference Only*). In Figure 4.9 this involves removing "at the goal" from the alignment input and relying only on the grammars knowledge about how `steal` (or "steals the ball") relates to `defense` (or *good defense by*) to make an entailment decision. This only slightly reduced the accuracy, which suggests that the real strength of the grammar lies in its knowledge of lexical relationships or lexical-based inferences.

Qualitative Analysis and Discussion

One benefit of our grammar-based approach is that we can inspect how the system reasons by looking at example derivation trees generated by the model. While our experiments show that the inference grammar does not increase accuracy on the original semantic parsing task, a manual inspection indicates that the model is

nonetheless learning improved representations, as we shown in Figure 4.14. In example 1, for example, the parser learns after being trained on the inference data that the modifier *under pressure* should be treated as a separate constituent in the parse tree. The particular analysis also captures the correct semantics by treating this phrase as forward-entailing, which allows us to predict how the entailment changes if we insert or delete this constituent. Similarly, the parser learns a more fine-grained analysis for the phrase *passes out* to by treating *out* as a type of modifier that does not affect entailment.

Figure 4.15 shows the types of knowledge learned by our system and used in proofs. The top row shows example *construction-based* inferences, or modifier constructions. For example, the first example treats the word *beautiful* in *a beautiful pass* as a type of modifier that changes the entailment or implication when it is inserted (forward-entails) or deleted (reverse-entails). In set-theoretic terms, this rule says that the set of *beautiful passes* is a subset of the set of all *passes*. The bottom row show types of *lexical-based* inferences, or relations between specific symbols. For example, the model learns that the `pink team` is disjoint from a particular player from the purple team, `purple9`, and that a `bad pass` implies a `turnover` event.

Figure 4.16 shows three common cases where our system fails. The first error (1) involves a sense error, where the system treats *shoots* as having a distinct sense from *shoots for the goal*. This can be explained by observing that *shoots* is used ambiguously throughout the corpus to refer to both shooting for the goal and ordinary kicking. The second example (2) shows how errors in the semantic parser (which is used to generate an alignment) propagate up the processing pipeline. In this case, the semantic parser erroneously predicted that *pink6* is the first argument of the `steal` relation (a common type of word-order error), and subsequently aligned *purple 8* to *pink6*. Similarly, the semantic parse tree for the hypothesis in the last (3) failed to predict *another* as a modifier, which would generate an alignment with the empty string λ. The last two cases show the limitation of our approach to generating alignments, and suggests that allowing the model to reason about different possible alignments might help avoid these errors.

Looking ahead, we note that while we use a simplified version of the full natural logic calculus (owing to the simplicity of the Sportscaster domain), our general

approach is amendable to more complex logical systems. For example, we could implement complex projection rules for quantifiers and other linguistic operators by simply introducing new symbols in our grammar with specially designed join rules. In introducing more complex rules that go beyond simple joins, however, we would be greatly expanding the space of possible proofs; whether learning in such a large space (with only minimal background assumptions) is feasible is an empirical question. It also remains to be seen whether using entailment as a learning signal might help to learn other types of complex linguistic structure, which is a question that we leave for future work.

4.7 Conclusions

In this chapter, we considered the problem of training semantic parsers in domains where the target logical forms are underspecified and fail to capture basic facts about entailment and inference. As a general solution to this problem, we introduced a new learning framework called *learning from entailment* that involves adding pairs of sentences annotated with entailment information to the semantic parser's training data. With this added data, we then force the semantic parser to generate explanations of the provided entailments, which forces the model to learn more about the target domain and find holes in the provided annotations.

To experiment with this idea, we performed experiments on the benchmark Sportscaster corpus from Chen and Mooney (2008a), which we expanded to include a corpus of sentence pairs annotated with entailments. As a way of operationalizing this idea of forcing the semantic parser to generate explanations, we created a novel grammar-based semantic parsing architecture and learning strategy that includes a probabilistic reasoning component based on the natural logic calculus (MacCartney and Manning, 2009). With this added machinery, the resulting model is able to jointly learn to generate logic forms, as well as perform symbolic reasoning over symbols in the target learning and solve entailment tasks.

Using our general approach, we achieved state-of-the-art results on the original Sportscaster semantic parsing task. To demonstrate the effectiveness of our model on modeling entailment, we also introduced a new RTE-style evaluation task based on the extended Sportscaster corpus, on which our main inference model strongly

outperformed several strong baselines (with around 73% accuracy). In conclusion, we found that learning from entailment can be an effective technique for improving the representations being learned for semantic parsing, and for making the learning of semantic parsers more robust.

5 Function Signature Semantics and Code Retrieval

> *"I like the direction of this work, and I definitely think that there is potential for an interesting semantic correspondence dataset in the relationship between code and its documentation, but I was a little disappointed that the dataset was essentially just (javadoc, function prototype) pairs. With that data only, I don't see how it be used for building a model that has a practical application..."*
>
> – Reviewer #3, *ACL 2017 Blind Reviews*

In this chapter, we briefly consider the semantics of the function signature representations being learned in Chapters 2-3. The general idea is to define a formal semantics for these function signature representations in terms of classical logic, as first proposed in Richardson (2018). We also discuss the application of our models to the task of code retrieval and API question-answering, and provide a system description of the `Function Assistant` and `Zubr` tool from Richardson and Kuhn (2017a). While the first two chapters of this thesis focus exclusively on the task of text to code translation (which we assume is a necessary first step towards deeper NLU for source code), the aim of this chapter is to describe more general applications and example uses of these datasets.

5.1 The Semantics of Function Signatures

When learning the component representations introduced in Chapter 2 (for examples, see Figure 5.1), one natural question to ask is: what do these representations actually mean? Unlike in conventional semantic parsing tasks, the function signature representations lack a well-defined semantics and other features such as compositionality. This is not altogether problematic in our early experiments, since our main idea is to use these representations (which resemble atomic predicate logic representations) to study the more general translation and lexicon learning

Docstring	Returns the greater of two long values
Signature	(Java) `lang Math long max(long a,long b)`
Docstring	Compares two values numerically and returns the maximum
Signature	(Python) `decimal Context max(a b)`
Docstring	gibt den größeren dieser Werte zurück
Signature	(PHP) `mixed max(mixed $value1, mixed $value2, ..)`

Figure 5.1: Example function docstring and signature pairs from Chapter 2 (in a conventionalized format) for the `max` function.

problem encountered in semantic parsing. Nonetheless, we can further formalize the signature representations being learned and define a semantics for the resulting language.

Under this approach, we define a simple domain-specific language and unified syntax for function signature representations across different programming languages and software projects, as well as a systematic mapping from this language into first-order logic and a small subject domain model (for a similar idea, see Bos (2016), see also Appendix C.1 for a technical overview of the logical notation used here). By recasting the learned representations in terms of classical logic, we aim to broaden the applicability of existing code datasets to studying more complex natural language understanding and reasoning problems in the software domain. In what follows, we define this general syntax and translation, and discuss the various applications that motivate our particular approach and subject domain model.

5.1.1 A Unified Syntax for Function Signatures

Definition 1. (Syntax of Function Signatures)

$\texttt{Signature} ::= \texttt{l N C} :: \texttt{f}(\texttt{t}_1\texttt{:p}_1, ..., \texttt{t}_n\texttt{:p}_n)\ \texttt{-> r}$

As discussed in Section 3.3, function signature representations across different programming languages consist of the following components: a *namespace* `N` (indicating the position or path in the target API), a *class* or *local name identifier* `C`, a *function name* `f`, a sequence of (optionally typed `t`) *named parameters* `p`, and an (optional) *return value* `r`. Below shows the different parts of the Java `max` function

first encountered in Figure 2.1:

$$\underbrace{\texttt{lang}}_{\texttt{N}}\ \underbrace{\texttt{Math}}_{\texttt{C}}\ \underbrace{\texttt{long}}_{\texttt{r}}\ \underbrace{\texttt{max}}_{\texttt{f}}\texttt{(}\ \underbrace{\texttt{long a, long b}}_{\texttt{t}_1\ \texttt{p}_1,\ \texttt{t}_2,\ \texttt{p}_2}\ \texttt{)}$$

In languages or software projects where some of this information is missing, we can mark the positions using special tokens, such as `UNK`, or unknown, for types in dynamically typed languages, or `core` and `builtin` in cases where the namespace and class information are missing. In Definition 1, we define a generic syntax for function signature representations in order to eliminate superficial differences between different programming languages. This definition includes an additional token `l` that identifies the particular programming language or software project from which the function `f` is drawn (see Figure 5.2 for a *normalized* version of our Java example).

5.1.2 Semantics and Translation to Logic

In order to provide a model theoretic semantics of these signature representations, we define a systematic mapping from `Signature` to logic. We also use a small inventory of domain specific predicates to define the semantics, which are motivated by some of the applications that we discuss in the concluding subsection.

Definition 2. (**Function Semantics**)
$[\![\texttt{l N C::f(t}_1\texttt{:p}_1\texttt{,...,t}_n\texttt{:p}_n\texttt{) -> r}]\!] =$
$\lambda x_1..\lambda x_n \exists v \exists f\ \texttt{fun}(f, \texttt{f}) \wedge \texttt{eq}(v, \texttt{f}(x_1 \ldots x_n)) \wedge \texttt{lang}(f, \texttt{l}) \wedge \texttt{type}(v, \texttt{r}) \wedge$
$[\![\texttt{C}]\!] \wedge [\![\texttt{N}]\!] \wedge [\![\texttt{t}_1\texttt{:p}_1]\!] \wedge ... [\![\texttt{t}_n\texttt{:p}_n]\!]$

The semantics can be described in the following way: for a given function f with some set of function variables $x_1, .., x_n$ (bound here using lambda abstraction), there should exist a value v which is equal to (shown here using using a special predicate `eq`) the value that results when the particular function constant `fun` is applied to said variables. For example, the variable v in the following example (where lambda conversion is performed on the input 4L, 5L):

$$[\![\texttt{java lang Math::max(long:a,long:b) -> long}]\!](\texttt{4L})(\texttt{5L})$$

Returns the greater of two long values

Signature (informal)	`lang Math long max(long a,long b)`
Normalized	`java lang Math::max(long:a,long:b) -> long`
Expansion to Logic	$[\![$`java lang Math::max(long:a,long:b) -> long`$]\!]$ $\Updownarrow$ $\lambda x_1 \lambda x_2 \exists v \exists f \exists n \exists c$ `eq`$(v,$`max`$(x_1,x_2)) \wedge$ `fun`$(f,$`max`$) \wedge$ `type`$(v,$`long`$)$ $\wedge$ `lang`$(f,$`java`$)$ $\wedge$ `var`$(x_1,$`a`$) \wedge$ `param`$(x_1,f,$`1`$) \wedge$ `type`$(x_1,$`long`$)$ $\wedge$ `var`$(x_2,$`b`$) \wedge$ `param`$(x_2,f,$`2`$) \wedge$ `type`$(x_2,$`long`$)$ $\wedge$ `namespace`$($n,`lang`$) \wedge$ `in_namespace`(f,n) $\wedge$ `class`$($c,`Math`$) \wedge$ `in_class`(f,c)

Figure 5.2: A normalized version of the Java example and its translation to logic.

takes the value of `5L`, or the result of applying `max`(4L,5L). In order to capture additional constraints about typing, naming, and the language from which the function is drawn, we use the following domain specific predicates: `fun` (associates the function variable f with the function constant or name `f`, e.g., `max`), `lang` (the language or project associated with f), and `type` (the type of a given variable, in this case relating the function return variable v with the return type constant `r`, e.g., `long`).

Arguments Definition 3 shows the semantics of function arguments.

Definition 3. (**Argument Semantics**)
$[\![\mathtt{t}_j\,\mathtt{:p}_j]\!] = \mathtt{var}(x_j, \mathtt{p}_j) \wedge \mathtt{type}(x_j, \mathtt{t}_j) \wedge \mathtt{has\_param}(f, x_j, \mathtt{j})$

The same naming and typing constraints are expressed using similar predicates for variables. The predicate `var` associates a given variable assignment x_j with an argument name $\mathtt{p}_j$. In addition, the predicate `has_param` explicitly associates a given argument or parameter and its position with a function f.

Namespace and Classes Definition 4 shows the semantics of namespaces and classes (or local names):

Definition 4. (**Namespace and Class Semantics**)
$[\![\texttt{N}]\!] = \exists n.\texttt{namespace}(n, \texttt{N}) \wedge \texttt{in\_namespace}(f, \texttt{n})$
$[\![\texttt{C}]\!] = \exists c.\texttt{class}(c, \texttt{C}) \wedge \texttt{in\_class}(f, \texttt{c})$

Here, we use the predicates `namespace` and `class` to identify the type of the variables n and c. As with arguments, two additional predicates, `in_namespace` and `in_class`, are introduced in order to associate particular namespaces and classes with particular function values.

Figure 5.2 shows a full translation from an ad hoc signature representation to a normalized representation and finally to a representation in logic using the definitions introduced above. We note that while we use a specific, and seemingly arbitrary, set of domain predicates, new predicates and information can be added as needed. In the next subsubsection, we motivate the particular predicates chosen above by describing some possible applications of our formulation.

5.1.3 Applications of the Logical Approach

In any application of logic, logical formulas can be used either to reason *extensionally* (i.e., about the particular real-world entities denoted by or involved in a given formula) or *intensionally* (i.e., about abstract relationships and consequences between concepts). Taking the example in Figure 5.2 and its expansion to logic, we could reason extensionally using pure logic about the exact value that this function will return given a particular input. In contrast, we could also, with the help of additional domain specific knowledge, reason intensionally about abstract relationships between different programming languages, class and namespace structures, and so on.

While we think that there is value in the first type of reasoning, especially for building executable models of functions, our primary focus is on reasoning abstractly about programming language constructs and relationships across different programming languages and projects. One benefit of the source code domain (as described in Chapter 2) is that much of the declarative knowledge needed for reasoning can be extracted straightforwardly from the target libraries directly, including information about class containment and subsumption relations, lists of related utilities (e.g., via *see-also* annotations and documentation hyperlinking), function

1.	Source (***en*, Haskell)**	**Input**: Shift the argument left by the specified number of bits.
Output	Language: **Haskell**	Trans: `Haskell Data.Bits builtin::shiftL(UNK:a,Int:UNK) -> UNK`
	Language: **Java**	Trans.: `Java java.math BigInteger::shiftLeft(int:n) -> BigInteger`
	Language: **Clojure**	Trans: `Clojure clojure.core builtin::bit-shift-left(UNK:x,UNK:n) -> UNK`

Figure 5.3: Polyglot translation output (in a normalized form) for the input *Shift the argument left by the specified number of bits.*

naming alternatives or aliases, and the relative position or distance between different functions and namespaces. Having such knowledge and an expressive logical language can in general facilitate more complex forms of API question-answering and code retrieval (a topic we discuss in the next section). As an example, we might might use the following notation (in which each `v?` expands to an existential variable in Definition 2):

```
java N? C?::f?(long:a,long:p?) -> long
```

to request the following: *Find some java function somewhere (i.e., in some class and namespace), that takes two long values as arguments (with the first value having the name `a`) and returns a long value.* Such a request might be used for finding structurally related functions or for mining *software clones* (Rattan et al., 2013).

Our primary focus is on building models that can robustly translate high-level natural language descriptions to code, and hence to the logical representations proposed above. We believe that under this scenario, natural language can prove to be a useful tool for deriving new forms of declarative knowledge. In Chapter 3, we looked at *polyglot translation*, where our semantic parsers translate descriptions into multiple programming languages. An example is provided in Figure 5.3, where the model translates the description about bit-shifting operations (originally drawn from the `Haskell` standard library) to equivalent function translations in the `Haskell`, `Java` and `Clojure` standard libraries. With this output, one could straightforwardly extract rules about function equivalences in different languages (e.g., `bit-shift-left` in `Clojure` is the same function as `shiftLeft` in `Java`), and learn further relationships between the associated function names and variables.

Using the notation introduced above, we can express cross language queries about equivalent functions in the following way:

```
java java.math BigInteger::EquivIn(
shiftLeft,haskell)(long:a, long:b) -> long
```

where the special predicate `EquivIn` is used to request the `Haskell` equivalent of the `shiftLeft` function in `Java`. The semantics of `EquivIn` can therefore be defined in the following way (where background knowledge about the `eq` predicate can be derived from the output of our polyglot model as discussed above):

$$[\![\texttt{l N C::EquivIn(f,lang)(t}_1\texttt{:p}_1\texttt{,...,t}_n\texttt{:p}_n\texttt{) -> r}]\!]$$
$$\Updownarrow$$
$$[\![\texttt{l N C::f(t}_1\texttt{:p}_1\texttt{,...,t}_n\texttt{:p}_n\texttt{)-> r}]\!] \wedge [\![\texttt{ lang N? C?::f'?(?)-> r? }]\!] \wedge \texttt{eq(f,f')}$$

One interesting direction is using general knowledge about software libraries and logic reasoning to help learn more robust translation models. The formalism introduced above is part of an effort to move in this direction, and we hope that integrating symbolic reasoning more generally will open the doors to new ideas and approaches.

5.2 Question Answering and Code Retrieval

As discussed above, one natural application of having a robust text to code semantic parser is question-answering over target software libraries or API. To some extent, the evaluation introduced in Chapter 2 already simulates a kind of code retrieval system: given an input text (or specification) at test time, the goal is to find the formal signature representation, within the space of all valid functions, that matches the specification. In this section, we motivate this application and discuss a simple prototype code retrieval system called `Function Assistant`.

```
## from nltk.parse.dependencygraph.py

class DependencyGraph(object):
   """A container ....for  a dependency structure"""

   def remove_by_address(self, address):
     """
     Removes the node with the given address.
     """
      # => implementation

   def add_arc(self, head_address, mod_address):
     """Adds an arc from the node specified by
     head_address to the node specified by
     the mod address....
     """
```

Figure 5.4: Example function documentation in Python NLTK about dependency graphs.

5.2.1 Motivation

Software developers frequently shift between using different third-party software libraries, or APIs, when developing new applications. Much of the development time is dedicated to understanding the structure of these APIs, figuring out where the target functionality lies, and learning about the peculiarities of how such software is structured or how naming conventions work. When the target API is large, finding the desired functionality can be a formidable and time consuming task. Often developers resort to resources like Google or StackOverflow to find (usually indirect) answers to questions.

We illustrate these issues in Figure 5.4 using two example functions from the well-known NLTK toolkit. Each function is paired with a short *docstring*, i.e., the quoted description under each function, which provides a user of the software a description of what the function does. While understanding the documentation and code requires technical knowledge of dependency parsing and graphs, even with such knowledge, the function naming conventions are rather arbitrary. As an example, the function:

```
add_arc(self,head_address,mod_address)
```

could just as well be called `create_arc`. An end-user expecting another naming convention might be left astray when searching for this functionality. Similarly, the available description might deviate from how an end-user would describe such functionality. Understanding the `remove_by_address` function, in contrast, requires knowing the details of the particular `DependencyGraph` implementation being used. Nonetheless, the function corresponds to the standard operation of *removing a node* from a dependency graph. Here, the technical details about how this removal is specific to a *given address* might obfuscate the overall purpose of the function, making it hard to find or understand.

At a first approximation, navigating a given API requires knowing *correspondences* between textual descriptions and source code representations. For example, knowing the following English descriptions corresponds (somewhat arbitrarily) to the following code fragments in Figure 5.4:

$$\textit{Adds an arc} \Rightarrow \texttt{add\_arc}, \textit{address} \Rightarrow \texttt{address}$$

One must also know how to detect paraphrases of certain target entities or actions, for example that *adding an arc* means the same as *creating an arc* in this context. Other technical correspondences, such as the relation between an `address` and the target dependency graph implementation, must be learned.

In the next subsection, we provide a system description of a code retrieval prototype we built called `FunctionAssistant`. Similar to the experiments in Chapters 2-3, the tool works in the following way: given a target API, we learn a MT-based semantic parser that translates text to code representations in the API. End-users can formulate natural language queries to the background API, which our model will translate into candidate function representations with the goal of finding the desired functionality. In this section, we focus on using the tool as a blackbox pipeline for building directly from arbitrary API collections.

5.2.2 Function Assistant Tool

The `Function Assistant` tool is a prototype code retrieval engine that natively implements all of the models introduced in Chapter 2. The tool is part of the companion semantic parsing toolkit **`Zubr`** that was used to implement virtually

```
## pipeline parameters
params = [
  ("-baseline","baseline",False,"bool",
 "Use baseline model [default=False]","GPipeline")
]

## Zubr pipeline tasks
tasks = [
  "zubr.doc_extractor.DocExtractor",# extract docs
  "process_data", # custom function.
  "zubr.SymmetricAlignment",# learn trans. model.
  "zubr.Dataset", # build dataset obj.
  "zubr.FeatureExtractor",  ## build extractor obj.
  "zubr.Optimizer", ## train reranking model
  "zubr.QueryInterface", # build query interface
  "zubr.web.QueryServer", # launch HTTP server
]

def process_data(config):
    """Preprocess the extracted data using a custom
     function or outside library (e.g., nltk)

     :param config: The global configuration
     """
     preprocess_function(config,...)
```

Figure 5.5: An example pipeline script for building a parallel API dataset, translation model and query server.

all of the content in this thesis. For efficiency, the core functionality is written in Cython (Behnel et al., 2011)[4], which is a compiled superset of the Python language that facilitates native C and C++ integration. `Function Assistant` is designed to be used in one of two ways: first, as a black-box pipeline to build custom translation pipelines and API query engines. The tool can also be integrated with other components using our Cython and Python API. We focus on the first type of functionality.

Library Design and Pipelines

The underlying `Zubr` library uses dependency-injection OOP design principles. All of the core components are implemented as wholly independent classes, each of which has a number of associated configuration values. These components interact

[4]for more information, see `http://cython.org/`

via a class called `Pipeline`, which glues together various user-specified components and dependencies, and builds a global configuration from these components. Subsequent instantiation and sharing of objects is dictated, or *injected*, by these global configurations settings, which can change dynamically throughout a pipeline run.

Pipelines instances are created by writing pipeline scripts, such as the one shown in Figure 5.5 (a more general `Zubr` pipeline for training neural models is shown in Figure 5.6). This file is an ordinary Python file, with two mandatory variables. The first `params` variable specifies various high-level configuration parameters associated with the pipeline. In this case, there is a setting `--baseline`, which can be evoked to run a baseline experiment, and will effect the subsequent processing pipeline. The second, and most important, variable is called `tasks`, and this specifies an ordering of subprocesses that should be executed. The fields in this list are pointers to either core utilities in the underlying `Zubr` toolkit (each with the prefix `zubr.`), or user defined functions.

This particular pipeline starts by building a dataset from a user specified source code repository, using `DocExtractor`, then builds a symmetric translation model `SymmetricAlignment`, a feature extractor `FeatureExtractor`, a discriminative reranker `Optimizer`, all via various intermediate steps. It finishes by building a query interface and query server, `QueryInterface` and `QueryServer`, which can then be used for querying the input API. Our current `DocExtractor` implementation supports building parallel datasets from raw Python source code collections. Internally, the tool reads source code using the abstract syntax tree utility, `ast`[5], in the Python standard library, and extracts sets of function and description pairs. In addition, the tool also extracts class descriptions, parameter and return value descriptions, and information about the API's internal class hierarchy. This last type of information is then used to define document-level features.

As noted already, each subprocesses has a number of associated configuration settings, which are joined into a global configuration object by the `Pipeline` instance. For the translation model, settings include, for example, the type of translation model to use, the number of iterations to use when training models, and so on. All of these settings can be specified on the terminal, or in a separate configuration file. As well, the user is free to define custom functions, such as `process_data`, or

[5] `https://docs.python.org/2.7/library/ast.html`

```
params = [
    ("–wdir","wdir","","str",
      "The working directory [default=']","NeuralRunner"),
     ("–name","name","","str",
      "The name of the dataset [default=']","NeuralRunner"),
     ("–make_subword","make_subword",False,"bool",
      "subword representations for english side [default=False]","NeuralRunner"),
       ...
]

description = {"NeuralRunner" : "settings for running neural model"}

tasks = [
  "setup_data", ## create initial data for Seq2Seq taining
  "zubr.wrapper.foma", ## build graphs
  "zubr.neural.run" ## train and run neural decoder
]

def setup_data(config):
      ...
```

Figure 5.6: An example pipeline for running the neural models from Chapter 3.

classes which can be used to modify the default processing pipeline or implement new or additional machine learning features.

Web Server The last step in this pipeline builds an HTTP web server that can be used to query the input API. Internally, the server makes calls to the trained translation model and discriminative reranker, which takes user queries and attempts to translate them into API function representations. These candidate translations are then returned to the user as potential answers to the query. Depending on the outcome, the user can either rephrase his/her question if the target function is not found, or look closer at the implementation by linking to the function's source code.

An example screen shot of the query server is shown in Figure 3. Here, the background API is the NLTK toolkit, and the query is:

Query: *Train a sequence tagger model*

While not mentioned explicitly, the model returns the method `train(...)` and other related methods for the `HiddenMarkovModelTagger`, thus inferring that a

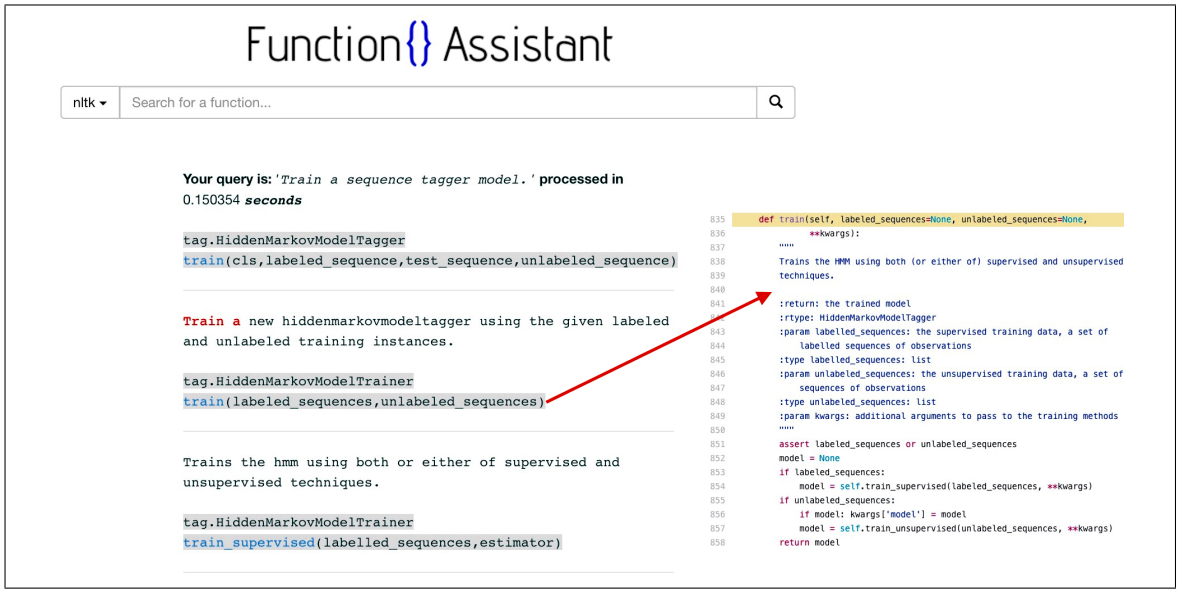


Figure 5.7: An example screen shot of the `Function Assistant` web server.

hidden markov model is a type of *sequence tagger*. The right side of the image shows the hyperlink path to the original source in Github for the `train` function.

Deployment and Future Work A public version of the `Function Assistant` web server has been running at `http://zubr.ims.uni-stuttgart.de/` with a number of example library models since September 2017. These models were trained on the datasets introduced in Chapter 2. As such, they only simulate potential user queries, and more work needs to be done to evaluate the collected queries and determine the accuracy of finding the correct methods.

The current version of `Function Assistant` does not yet include the polyglot features discussed in Chapter 3, as well as the mapping to logic described at the beginning of this chapter. One future direction is to provide this additional functionality, and provide a reasoning mechanism that can be evoked to reason across APIs using abstract knowledge about the underlying structure of the provided libraries. More work should be done also to allow for more abstract types of queries (e.g., querying about generic sets of functions, or abstracting over variables).

5.3 Conclusions

In this chapter, we consider several applications of the source code datasets introduced in Chapters 2-3, largely centering around different types of automated question-answering. Our proposal to define a formal semantics of function signatures in terms of classical logic aims to broaden the scope of how these representations can be used to reason more generally about API libraries and programming constructs. The use of logical representations also aligns our general with more conventional semantic parsing tasks and the NLU approach introduced in Chapter 1. We also discuss an initial attempt at building and deploying a code retrieval prototype called `Function Assistant` that allows for building models on arbitrary source code projects.

Moving ahead, we see a lot of potential in code retrieval and API question-answering as an application of the ideas pursued in this thesis. While our current approach and implementation uses simple components, we hope that our tools and ideas will serve as a benchmark for future work in this area, and ultimately help to solve everyday software search and reusability issues.

6 Thesis Conclusions

In this chapter, we provide a brief summary of the contributions made in this thesis, and discuss some high-level technical themes that emerged from the different chapters. We end by reflecting on the overall program for natural language understanding (NLU) first introduced in Chapter 1, and describe what we see as some of the challenges that lie ahead.

6.1 Thesis Contributions

The ultimate goal of this thesis is to develop new techniques and resources for building robust semantic parsing models that require minimal amounts of hand engineering. In doing this, we tackled several of the underlying resource problems that one inevitably encounters when trying to do this (as described at the beginning of each chapter). In this section, we again summarize each problem and our proposed solutions and contributions.

Source Code as Parallel Data (Chapters 2,5) The most fundamental resource problem for data-driven semantic parsing is finding the parallel data needed to train the underlying models given that such data does not naturally occur *in the wild*. To get around this, we proposed using automatically extracted source code documentation, or collections of text and formal code representations, as a parallel corpus for benchmarking and developing new semantic parsing models. We introduced 45 new datasets in this domain that span a wide range of natural languages and programming languages, as well as a new and challenging text-to-code semantic parsing task and code retrieval application.

On this new task, we encountered several technical challenges, largely related to the broad scope of the target datasets and the sparsity of the formal representations being learned, which made it hard to apply standard semantic parsing techniques. This required us to develop several new baseline models, based primarily on statistical machine translation (SMT) techniques, and new decoding strategies that take into account the structure of the output languages being generated (which is

a topic that we refer to throughout the thesis as *constrained decoding*; more about this below).

Training on Multiple Datasets (Chapter 3) Even when parallel data is available, the amount of domain- or language-specific data might still be insufficient for learning a robust model. To address this issue, we looked at learning semantic parsers from multiple datasets, and building *polyglot* models that can translate between arbitrary language pairs (including language pairs not observed during training). To facilitate modeling of this kind, we develop a novel graph-based decoding framework and experimented with several types of translation models that work within this framework.

Using polyglot modeling, we improved our initial results on the technical documentation datasets, and achieved results competitive with the state-of-the-art of two additional benchmark tasks. We also introduced a novel *mixed language* semantic parsing task that highlights the benefits of polyglot modeling versus monolingual modeling (i.e., training models over a single language or domain).

Learning to Make Inferences (Chapter 4) Even when the first two resource problems have been solved, the formal representations provided in a given dataset might fail to capture all aspects of language meaning. To deal with this issue, we developed a new learning framework called *learning from entailment* that uses entailment information (i.e., high-level inferences about whether the meaning of one sentence follows from another) as a weak learning signal to train semantic parsers to reason about the holes in their analysis and learn improved semantic representations.

To accomplish this, we developed a new semantic parsing and reasoning model that jointly learns from logical forms and sentence pairs annotated with entailment information. With this model, we achieve state-of-the-art results on one benchmark semantic parsing tasks, and introduced a new entailment task evaluation for measuring the ability of semantic parsers to model inference and reasoning.

Discussion: The Need for Better Evaluation

Beyond using the new ideas and resources described above to improve on standard tasks, we were also motivated by a need for more diverse task evaluations in semantic parsing. There is a common tendency to evaluate semantic parsing intrinsically in terms of translation accuracy, which largely concerns determining whether a generated representation is correct or not. One theme throughout the thesis is using more complex extrinsic tasks to evaluate model performance, which focus more on the *the types of complex problems* that the representations being learned help to solve. For example, our use of mixed language and entailment testing involves seeing whether a semantic parser's output representations can be used to effectively model code-mixing and simple types of symbolic reasoning.

Moving forward, we still see a large need for developing more diverse task evaluations, especially ones involving entailment and inference modeling, which despite being the *raison d'être* of semantics, has received little attention in the literature. As discussed in Chapter 5, we think that our software datasets could prove to be a valuable resource for experimenting with end-to-end language understanding given their broad scope and highly structured nature. While they have the benefit of being automatically extracted, therefore not requiring manual annotation of formal representations, a remaining question is whether the target code representations are sufficient for modeling deeper types of natural language semantics.

Discussion: Technical Themes and Structured Decoding

To investigate the different topics outlined above, we experimented with several different types of probabilistic semantic parsing models, ranging from models based on classical statistical machine translation (SMT, Chapters 2-3), conditional log-linear models (Chapter 2) and probabilistic grammars (Chapter 4) to more recent neural translation models (Chapter 3). In all cases, the modeling objective is to learn some conditional probability distribution $p(\mathbf{z} \mid \mathbf{x})$ of semantic outputs $\mathbf{z} \in \mathcal{L}_{out}$ given text input $\mathbf{x}$, and the decoding problem involves solving the following (given a particular input $\mathbf{x}$):

$$\mathbf{z}^* = \arg\max_{z \in \mathcal{L}_{out}} p(\mathbf{z} \mid \mathbf{x})$$

In solving any such decoding problem, a natural question to ask is: what assumptions can we make about the scope of the arg max and $\mathcal{L}_{out}$? One fundamental difference between semantic parsing and ordinary translation is that the former involves translating to formal languages, which allows us to make strong assumptions about the structure of $\mathcal{L}_{out}$. The topic of how one does decoding in a way that takes advantage of this structure, which we have referred to throughout the thesis as *constrained decoding*, is therefore a core topic in semantic parsing and one that has recently attracted interest in the context of neural semantic parsing (Krishnamurthy et al., 2017; Yin and Neubig, 2017; Rabinovich et al., 2017), where standard formulations of decoding assume a rather unconstrained search space.

A unifying feature of all the models we considered is the use of (labeled) directed graph representations to *constrain* and represent the output translation space and $\mathcal{L}_{out}$ (where paths in these graphs correspond to valid output expressions and labeled edges correspond to individual output words z) and the use of shortest-path search over these graphs as a way to efficiently solve the associated decoding problems. This includes our use of DAG representations in Chapter 3 for constraining SMT (see Algorithm 5) and neural (see Algorithm 6) decoding (what we call *shortest-path decoding*). In the general case, shortest-path search for DAGs involves the following procedure (see Algorithm 4 for more details):

0: $d[0] \leftarrow 0.0$
1: **for** each graph vertex $v \in V$ in sorted order
2: **do** $d(v) = \min_{(u,v,z) \in E} \Big\{ d(u) + w(u, v, z) \Big\}$

Under our translation approach, where we start with an input text $\mathbf{x}$ to be translated, the intuition is the following: the role of the edge weight function w is to assign a translation score to each edge label z conditioned on $\mathbf{x}$. Our main idea, therefore, is to replace w with translation models that dynamically generate edge weights during the ordinary search procedure. As we describe in some technical detail in Section 4.4, this idea has strong connections with classical hypergraph decoding for grammar-based models (see Algorithm 9), of the type we use in Chapter 4. Given these connections, we think that shortest-path decoding can provide a more general search framework for unifying the different approaches to semantic parsing, in particular grammar-based approaches and neural-based translation

approaches.

Beyond providing a high-level framework for describing and unifying different semantic parsing approaches, such a framework could also help to understand deeper theoretical questions about the complexity of constrained decoding for semantic parsing and provide a uniform toolset for helping to solve the underlying search-related problems.

6.2 Looking Ahead

> *"...language comprehension from an AI point of view assumes that language understanding depends on a lot of "real-world knowledge' and that our programs must have it if they are ultimately succeed. Fortunately, there is a branch of AI–knowledge representation–whose purpose in life is to provide this knowledge... thus we in AI-NLP go about our business not worried unduly by the fact that we do not actually have the knowledge base required by our most basic assumptions.... There is nothing wrong with this model as far as it goes. But at the same time anyone familiar with AI must realize that the study of knowledge representation... is not going anywhere fast. This subfield of AI has become notorious for the production of countless non-monotonic logics...none of the work shows any obvious application to actual knowledge representation problems... Thus many of us in AI-NLP have found ourselves in the position of basing our research on the successful completion of others' research–a completion that is looking more problematic. It is therefore time to switch paradigms...'*
>
> – Eugene Charniak (1996) *Statistical Natural Language Learning*

In this final section, we make some high-level remarks about the general program for data-driven natural language understanding introduced in Chapter 1 and assumed throughout the thesis. As emphasized from the beginning, our approach largely follows the classical symbolic approach (what Charniak above calls the *AI-NLP* approach), within which semantic parsing plays the vital role of translating natural language into symbolic representations that can be used for symbolic reasoning and program execution (we have elsewhere referred to this as the *compiler*

model for NLU). The novelty of recent work on data-driven semantic parsing is the attempt to learn such translations from data as way of getting around the difficulty of hand-engineering translation rules.

One problem with our general approach, however, is that solving the semantic parsing problem is only one step in solving the bigger language understanding problem, since the overall success of our research program also requires solving the associated knowledge representation and reasoning (KR&R) problems (i.e., problems associated with how we generally *use* formal representations to efficiently compute inference and solve real-world tasks, which involves a whole new set of bottlenecks and resource problems). It must be acknowledged, therefore, the ultimate success of our research program depends on the *successful completion of others' research* in KR&R, as Charniak discusses in the quote above.

For Charniak, this reliance on others' success is reason to abandon the AI-NLP program altogether. In this thesis, we take a middle ground and propose investigating the extent to which AI-NLP systems and theories can be modeled empirically and inferred from data using machine learning. As discussed in Chapter 1, Liang and Potts (2015) argue that recent developments in semantic parsing on learning 'compositional semantic theories from corpora and databases' are quickly *eroding* the divide between logical and statistical approaches to language, thus putting part of the AI-NLP approach within the purview of statistical modeling. Given the success of semantic parsing, we believe that other components in the end-to-end AI-NLP pipeline can benefit from machine learning in a similar fashion.

Moving forward, we see the following general research themes as important in the endeavor to build more robust end-to-end models:

- **More General and Robust Models**: As described in Chapter 1, the goal in most semantic parsing research is to build models that are *domain agonistic*, or not tied to particular domains and language types. We believe that we need more of this, especially related to building models that generalize well in low-resource or sparse settings, such as the technical documentation domain we consider in this thesis. What's currently missing is a systematic study into how semantic parsing models perform over a wide and diverse range of domains and knowledge representation types.

Given the success of neural sequence to sequence (Seq2Seq) modeling techniques in semantic parsing, we suspect that more general and robust Seq2Seq methods will need to be discovered to make this possible. However such models will likely need to respect the constrained nature of semantic parsing task and the peculiarities of translating to formal languages. Our work on *constrained decoding* assumes that we can make neural models more robust by supply them with more explicit information about the output prediction space, thus making them more like classical grammar-based models. The extent to which models need this information and can learn such structure is an open question.

- **Looking at Semantic Parsing Holistically** As discussed at the beginning, many semantic parsing evaluations are intrinsic in nature, and one theme throughout this thesis is to look at semantic parsing more holistically. For example, in Chapter 4 we observe the tight connection between reasoning and semantic parsing, and propose a model that jointly solve both tasks. This required us to consider the interface between these two different modeling components, and in the end led to a novel approach to doing KR&R.

  Therefore, we believe that having a systematic understanding of the relationship between semantic parsing and KR&R, as well as developing new tasks that evaluate on entailment and reasoning, will also be needed for building more robust natural language understanding models. We also think that by linking KR&R with learning semantic representations, we might discover new and interesting formalisms and learning techniques for data-driven KR&R.

A Code Datasets, Reranker Features, EM

A.1 Dataset Information

We report additional details about our dataset collection.

Standard Library Documentation Figure A.2 shows additional details of the different standard library datasets used in Chapter 2-3, including pointers to the original source of the documentation, the standard library version numbers, and other details. To our knowledge, none of these datasets, excluding the Java Standard Library set from Deng and Chrupała (2014), have been used for the types of NLP experiments that we describe in the thesis.

Technical Manuals We also show information about the Unix dataset first reported in Richardson and Kuhn (2014). While we introduced the dataset in Richardson and Kuhn (2014), Richardson and Kuhn (2017b) is our first attempt as using these datasets for semantic parsing.

A.2 Reranking Features

Figure A.3 shows the full feature set used by our reranking models.

Class information Some documentation sets include assertions about related functions or utilities, in the form of `see also` sections or links to other parts of the API (see examples in Figure A.1). Such information can also be found in quick reference manuals or language cheat sheets available online, as well as from html structure.This information is used to define features in our discriminative model.

Parameter descriptions In many datasets, the function documentation include additional textual descriptions of the function parameters. For the baseline trans-

Language	Example Classes
Ruby	{logger.info, logger.warn, logger.fatal, logger.debug, ... }
Elisp	{sin, cos, tan, asin, acos, atan, exp, log, log10, ... }
Unix	{iotop,iosnoop,iopattern,iopending, ... }
PHP	{ps_close_image, ps_open_image_file, ps_place_image, ... }
C	{UINT8_MAX UINT16_MAX UINT32_MAX UINT64_MAX UINT_FAST8_MAX, ... }

Figure A.1: Example classes, or abstract groupings of symbol types, extracted using document-level information in target APIs.

lation models, these can add these fragmented pairs to the parallel training data. We also use this information as features in our reranking model.

Return descriptions Similarly, some documentation also contains textual descriptions of return values, which can be used in the same way as described above.

Section descriptions Section descriptions are module or class level textual descriptions.

Feature Selection A greedy, backward search selection method was employed for some datasets where overfitting seemed to be an issue. This is done in the following way: after training a complete model, features or templates that lead to incorrect predictions on the validation are greedily removed, and those whose removal increases the accuracy on the validation are shut off. The model is then retrained used the resulting selected set of features or templates. More details are documented in our source code release.

A.3 Justification of EM Updates

As detailed in the chapter, the goal of the expectation maximization (EM) algorithm is to maximize the likelihood of the target training corpus D, where we can represent the likelihood of our parameters θ in the following way for our word-based

models:

$$\ell(\theta) = \log \prod_{d=1}^{|D|} p_\theta(\mathbf{x}^{(d)} \mid \mathbf{z}^{(d)}) \tag{A.1}$$

$$= \sum_{d=1}^{|D|} \log \sum_{a \in \mathcal{A}} p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \tag{A.2}$$

Recall also that the general form of the EM algorithms is the following:

$$\mathcal{Q}(\theta \mid \theta^t) = \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} \left[p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \right] \tag{A.3}$$

with which the E-step involves collecting the expected counts of alignments (i.e., the latent variables) using the posterior distribution $p_{\theta^t}(a \mid \mathbf{x}, \mathbf{z})$ and parameters θ^t, and the M-step involves finding new parameters θ that solve an $\arg\max_\theta$ over this equation.

As remarked in Bishop (2006), the general form of the expectation $\mathcal{Q}$ in Equation A.3 might seem arbitrary at first glance, and one question is whether this approach is guaranteed to increase our data likelihood after each iteration. In the general case, such a guarantee does hold, and proving this and deriving $\mathcal{Q}$ involves a result about concave functions called *Jensen's inequality* (Jensen, 1906). This result states that for any concave function f and set of variables x_j, λ_j s.t. $\sum_i^n \lambda_i = 1$, the following inequality holds (where equality holds when x_i takes the form $\frac{x_i}{\lambda_i}$ and λ_i is held constant):

$$f\Big(\sum_i^n \lambda_i x_i\Big) \geq \sum_i^n \lambda_i f(x_i) \qquad \text{Jensen's Inequality} \tag{A.4}$$

Given that $\log(\cdot)$ is concave function, the general idea is that the likelihood in Equation A.2 serves as the left side of the inequality, and the posterior in Equation A.3 plays the role of λ. In order to see this and to move the posterior, or λ, into

the likelihood computation, we can do the following:

$$\ell(\theta) = \sum_{d=1}^{|D|} \log \sum_{a \in \mathcal{A}} p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \tag{A.5}$$

$$= \sum_{d=1}^{|D|} \log \sum_{a \in \mathcal{A}} \frac{p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(i)})}{p_{\theta^t}(a \mid \mathbf{x}^{(i)}, \mathbf{z}^{(d)})} p_\theta(\mathbf{x}^{(i)}, a \mid \mathbf{z}^{(d)}) \qquad \text{Mult. by 1} \tag{A.6}$$

$$= \sum_{d=1}^{|D|} \log \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \frac{p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)})}{p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)})} \tag{A.7}$$

$$\geq \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log \frac{p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)})}{p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)})} \qquad \text{Jensen's Inequality} \tag{A.8}$$

$$\begin{aligned} = &\sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log p_\theta(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)}) \\ &- \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \end{aligned} \tag{A.9}$$

The first part of expanded fraction in Equation A.9 is equivalent to our expectation $\mathcal{Q}$. Given the second half of the equation does not involve θ and our objective is to find θ that maximizes the complete data likelihood, it suffices to only maximize the first part of the equation. Finally, to see that optimizing $\mathcal{Q}$ increases the data likelihood, we observe the following:

$$\ell(\theta^{t+1}) \geq \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log \frac{p_{\theta^{t+1}}(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)})}{p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)})} \qquad \text{Jensen's Ineq.}$$

$$\geq \sum_{d=1}^{|D|} \sum_{a \in \mathcal{A}} p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)}) \log \frac{p_{\theta^t}(\mathbf{x}^{(d)}, a \mid \mathbf{z}^{(d)})}{p_{\theta^t}(a \mid \mathbf{x}^{(d)}, \mathbf{z}^{(d)})} \qquad \theta^{t+1} \text{ is } \arg\max_\theta \mathcal{Q}$$

$$= \ell(\theta^t) \qquad \text{via equality constraint}$$

While these set of equations are specific to the word translation models investigated in Chapter 2, the general result holds for any such model with latent models. The main challenge, however, often involves finding a way to compute the posterior in $\mathcal{Q}$, which for some models does not have a closed-form solution.

Dataset	Version	Source(s)		Document Features		
				class info.	param.return section	
Java	SE 6.0	Deng and Chrupała (2014)	**main data**	see-also	✓	✓
Ruby	2.3.0	`ruby-doc.org/core-2.3.0/`	**main data**	fun.	✓	✗
		`ruby-doc.org/stdlib-2.3.0/`	**main data**	links		
PHP	3.0	`php.net/download-docs.php`	**main data**	see-also	✓	✓
Python	2.7.11	`docs.python.org/2.7/library/`	**main data**	✗	✗	✓
		`docs.python.org/2/reference/`	**background**	html	–	–
		`docs.python.org/2.7/library/` (*numpy*)	**num. library**	html	✓	✗
Elisp	25.1	`gnu.org/software/emacs/manual`	**main data**	✗	✓	✓
		`wikemacs.org/wiki/Emacs_Lisp_Cheat_Sheet`	**background**	html	–	–
		`github.com/magnars/s.el` (*s.el*)	**string library**	html	✗	✗
		`github.com/magnars/dash.el` (*dash.el*)	**list library**	html	✗	✗
Haskell	4.8.1	`hackage.haskell.org/package/base-4.8.1.0`	**main data**	✗	✗	✗
Clojure	1.7	`clojure.org/api/api`	**main data**	✗		
		`clojuredocs.org/clojure.org/api/cheatsheet`	**main data background**	see-also	✗	✗
		`github.com/weavejester/medley` (*medley*)	**fun. library**	see-also	✗	✗
		`github.com/Raynes/fs` (*fs*)	**file-sys library**	html	–	–
		`github.com/ztellman/gloss` (*gloss*)	**byte library**	✗	✗	✗
				✗	✗	✗
				✗	✗	✗
		`github.com/clj-time/clj-time` (*clj-time*)	**time library**	✗	✗	✗
C	2.24	`gnu.org/software/libc/manual/`	**main data**	✗	✗	✗
		`en.cppreference.com/w/c`	**main data**	html	✓	✓
MIT Scheme	9.2	`https://www.gnu.org/software/mit-scheme`	**main data**	html	✗	✗
Unix	–	Richardson and Kuhn (2014)	**main data**	see-also	✓	✗

Figure A.2: Further details of our corpus collection, including any background resources (**background**) or third party libraries (shown under double line) that were used. The last column shows additional document features used in our experiments. *Class* refers to information about general classes of functions, and *param* and *return* specify if the additional textual description of parameter values and return values (respectively) are included.

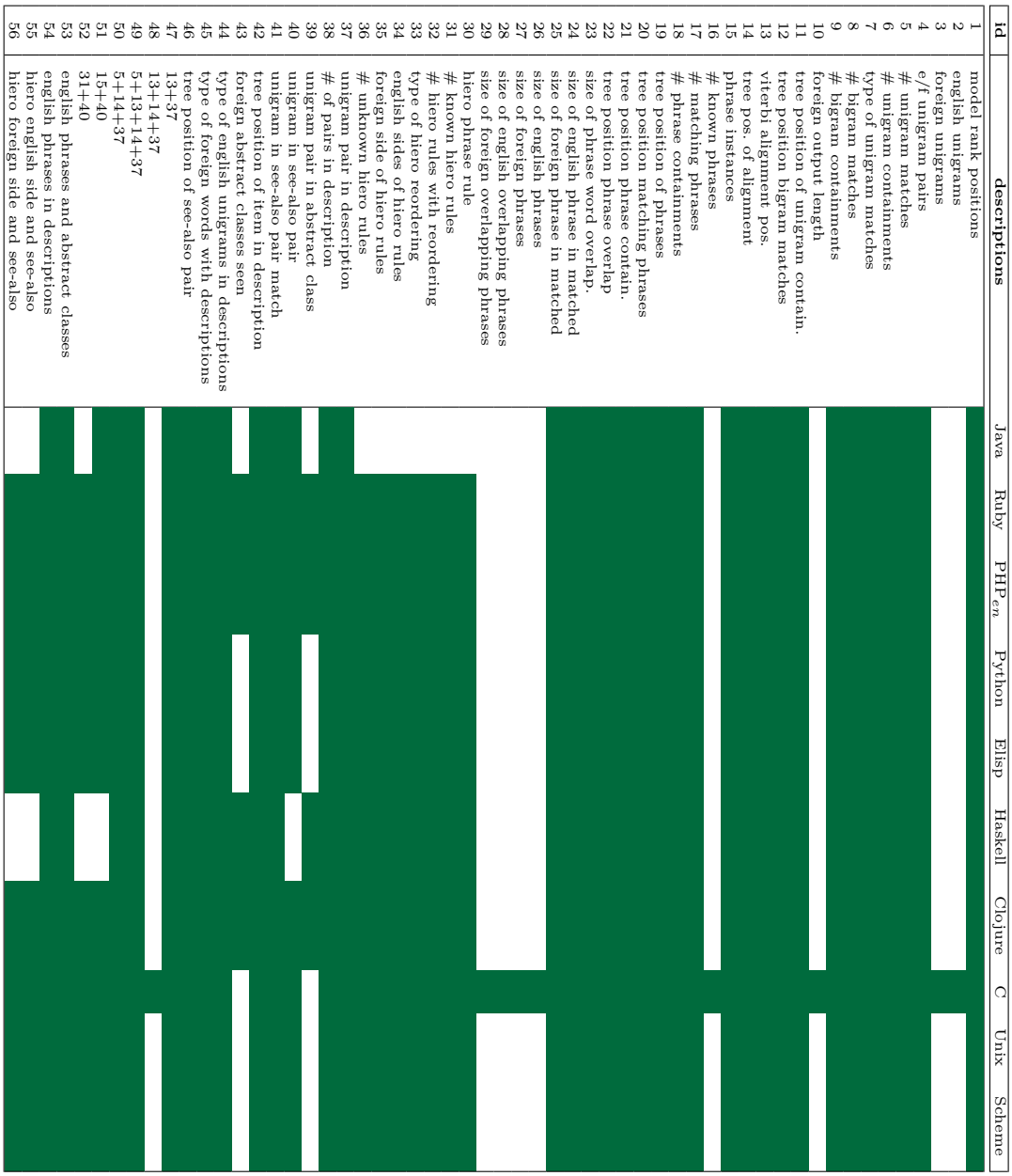


Figure A.3: A complete list of the features used in our reranker model, and an illustration (using green highlighting) of the features that were used in the best performing English models in the Stdlib experiments found during an ablation and feature selection study.

B Neural Network Primer, Graph Decoder Details

B.1 Neural Network Definitions

In this section, we give a brief overview of the neural network notation used in the Chapter 3, following the reviews from Goldberg (2016); Neubig (2017). Standardly, we use linear algebra notation where bold upper case letters denote matrices ($\mathbf{W}$) and bold lowercase letters ($\mathbf{b}$) denote vectors, with subscripts added accordingly to distinguish between different components. The vector $\mathbf{x}$ (using variations of the letter x) will be used throughout to denote an input vector, $\left[\mathbf{x}_1; \mathbf{x}_2\right]$ is used to denote vector concatenation, and **Wb** and **b + b'** will denote matrix-vector multiplication and vector addition, respectively.

B.1.1 Multi-layer Perceptrons

A 1-layer multi-layer perceptron (MLP) (see example in Figure B.1), can be defined in terms of the following operations (we will sometimes use multiple variables $\text{MLP}(\mathbf{x}_1, .., \mathbf{x}_2)$ which will implicitly denote vector concatenation of $\mathbf{x}_1, .., \mathbf{x}_n$):

$$\texttt{MLP}_1(\mathbf{x}) = \mathbf{W}_2(s(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_2 \tag{B.1}$$

where $s(\cdot)$ denotes a *non-linear activation function* (e.g., tanh), $\mathbf{x}$ is an input vector of dimension d (denoted as $\mathbf{x} \in \mathbb{R}^{|d|}$), $\mathbf{W}_1$ is a matrix with outer dimension d_2 ($\in \mathbb{R}^{|d_2| \times |d|}$), $\mathbf{W}_2$ is a second matrix $\in \mathbb{R}^{|d_3| \times |d_2|}$, and $\mathbf{b}_1$ and $\mathbf{b}_2$ are both vectors called *bias terms* ($\mathbf{b}_1 \in \mathbb{R}^{d_2}$,$\mathbf{b}_2 \in \mathbb{R}^{d_3}$). Using this basic form, more *layers* can be added as needed (to create *deeper* n-layer MLPs) by creating additional matrices, bias terms, and non-linear transformations. In contrast, by removing the non-linear function s and the hidden layer $\mathbf{W}_2$, the resulting model would be a linear model similar to the log-linear model used in Chapter 2.

When using these networks for classification, the final layer will have an outgoing

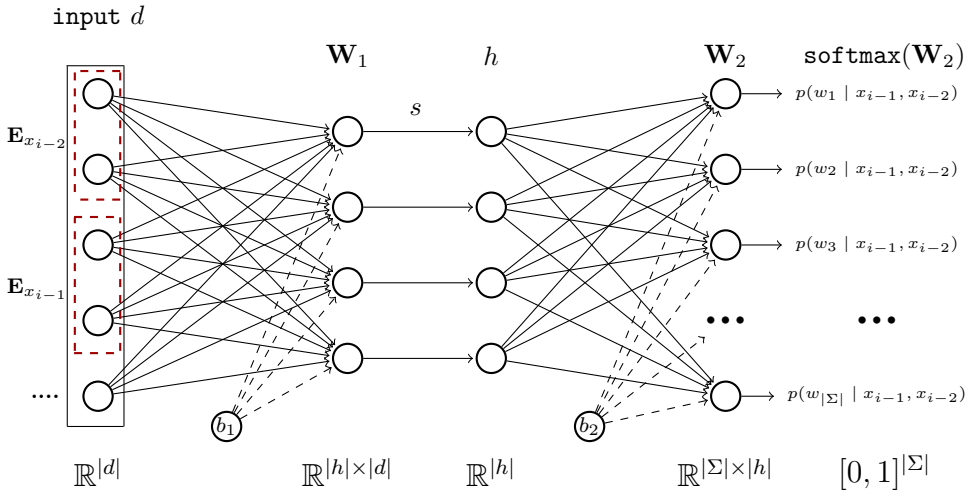


Figure B.1: An example multi-layer perceptron (MLP) network applied to language modeling.

dimensions equal to the size of the target classes. For example, Figure B.1 shows a MLP used for bigram language modeling, where the goal is to predict a new word given the previous two words within the class space of all possible output words (shown as Σ). In this scenario, one common output transformation is called *softmax* (a type of vector-to-vector transformation, also referred to as a *softmax layer*):

$$\texttt{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{B.2}$$

which in the end will create a discrete probability distribution over the output words or classes, with which we can find the most likely output class or compute the likelihood of some data.

One innovation in recent work on neural networks is the use of low-dimensional vector representations for discrete entities such as words (also referred to as *embeddings*). In the example above, for example, the previous words on the input side, x_{i-1}, x_{i-2}, are represented not as single *discrete* points in the input vector $\mathbf{x}$ (as we might have done using our log-linear model in Chapter 2), but as the concate-

nation of two vectors taken from some embedding matrix $\mathbf{E} \in \mathbb{R}^{|\Sigma_{in}| \times b}$ (with inner dimension b). By treating words as vectors, the subsequent vector representations (which are learned parameters in the model) allows the model to generalize across related words.

Learning Similar to the log-linear models covered in Chapter 2, we can train these models using stochastic gradient-descent, which will involve defining some loss function L, and solving the following gradients w.r.t. this loss function and individual network parameters: $\frac{\partial L}{\partial \mathbf{w}_1}, \frac{\partial L}{\partial \mathbf{w}_2}, \frac{\partial L}{\partial \mathbf{b}_1}, \frac{\partial L}{\partial \mathbf{b}_2}$,... The technique of *backpropogation* allows for solving this efficiently by working backwards from the output and exploiting the chain rule (for more details, see Goodfellow et al. (2016)).

B.1.2 Recurrent Neural Networks

Recurrent neural networks (RNN) are a generalization of the MLP model introduced above that allow for sequence modeling. In the most basic form, they work by feeding the output state of each previous input in a sequence $\mathbf{g}_{j-1}$ (starting from some initial state $\mathbf{g}_0$) when evaluating a new input $\mathbf{x}_j$, as given by the following recursive definition:

$$\mathbf{g}_j = \texttt{RNN}(\mathbf{g}_{j-1}, \mathbf{x}_j) \tag{B.3}$$

$$= s(\mathbf{W}_1 \mathbf{x}_j + \mathbf{W}_{1'} \mathbf{g}_{j-1} + \mathbf{b}) \tag{B.4}$$

where a probability distribution at time j (p_j) can be obtained (again, in the context of classification and model training) by doing the following:

$$p_j \sim \texttt{softmax}(\mathbf{W}_2 \mathbf{g}_j + \mathbf{b}_2) \tag{B.5}$$

One of the benefits of the RNN model is that the recurrent states allow for conditioning each decision on the full sequence history. For tasks such as language modeling, this eliminates the need to make Markov assumptions (i.e., deciding how far back in the sequence to look, as we did in the MLP example in Figure B.1). Training these models can also be done efficiently using a variant of backpropagation called *backpropagation through time* (Werbos, 1990) that involves backpropagating

a global loss for a given input sequence through all the time steps in that sequence.

The difficulty with vanilla RNNs, however, is that backpropagating through long sequences can sometimes have the effect of making the later gradients vanish, or become so small that they have little effect on the parameter updates (Pascanu et al., 2013). To deal with this *vanishing gradient* problem, several specialized architectures have been proposed, most notably variants of the Long Short-Term memory (LSTM) architecture of Hochreiter and Schmidhuber (1997). The basic idea is change the standard RNN architecture to include a memory cell c that preserves gradients over time and uses a set of *gating* mechanisms (analogous to logic gates) that determine how much of an given input to store in memory. The LSTM state $\mathbf{g}'$ is defined recursively in the following way:

$$\mathbf{g}'_j = \texttt{LSTM}(\mathbf{g}'_{j-1}, \mathbf{x}_j) \tag{B.6}$$

$$= \mathbf{o}_j \odot s(\mathbf{c}_j) \tag{B.7}$$

which in turn relies on the following set of equations (where $\odot$ denotes pointwise multiplication, and $\sigma(x)$(*the sigmoid* or *logistic* function) $= \frac{1}{1+e^{-x}}$):

$$\mathbf{c}_j = \mathbf{i}_j \odot \mathbf{u}_j + \mathbf{f}_j \odot \mathbf{c}_{j-1} \tag{B.8}$$

$$\mathbf{u}_j = s(\mathbf{W}_1 \mathbf{x}_j + \mathbf{W}_{1'} \mathbf{g}_{j-1} + \mathbf{b}_1) \tag{B.9}$$

$$\mathbf{o}_j = \sigma(\mathbf{W}_2 \mathbf{x}_j + \mathbf{W}_{2'} \mathbf{g}'_{j-1} + \mathbf{b}_2) \tag{B.10}$$

$$\mathbf{f}_j = \sigma(\mathbf{W}_3 \mathbf{x}_j + \mathbf{W}_{3'} \mathbf{g}'_{j-1} + \mathbf{b}_3) \tag{B.11}$$

$$\mathbf{i}_j = \sigma(\mathbf{W}_4 \mathbf{x}_j + \mathbf{W}_{4'} \mathbf{g}'_{j-1} + \mathbf{b}_4) \tag{B.12}$$

Here u_j in Equation B.9 is the same as the vanilla RNN in Equation B.15, and c_i, o_i, and f_j are input, output, and forget gates that either allow or block certain types of information. Intuitively, the sigmoid function will transform each vector value to a value between 0 and 1, such that values closer to 1 will *open* the gate and allow information to pass through, whereas values closer to 0 will *close* the gate and block information (for more details, see Neubig (2017)).

B.2 Graph Decoder

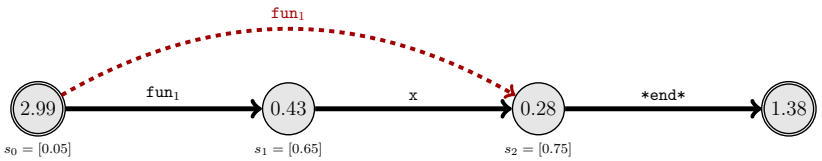
B.2.1 General Decoder Complexity

As detailed in the chapter, our decoder uses the k-shortest path procedure of Yen (1971). In the case of `DAG`s, this algorithm has a time complexity of $O(k\,|V|\,(|V|+|E|))$ for $k > 1$. This complexity can be explained in the following way: $O(|V|+|E|)$ (where V and E are the graph nodes and edges respectively) is the complexity of the `DAG` single shortest path procedure (or for $k = 1$). For each k, we consider l number of `new_start` positions in the most recent $k-1$ SSSP (starting on Line 5), which in the worst case can be of size $|V|$. Each branching path $j \in l$ then requires a run of the SSSP procedure of complexity $O(|V| + |E|)$ (as stated above).

In Algorithm 7, we show several optimizations that improve the runtime (though not the complexity) of the procedure, including starting each nested call to SP at `new_start` as opposed to searching through the full graph, and using a `min` heap to store candidate shortest path in Lines 2,13 and 15 as opposed to having to re-sort B each time at line 15. Another frequently used optimization trick (not shown here), known as Lawler's trick (Lawler, 1972), involves keeping track of already computed branching paths so as to avoid solving for duplicate candidates shortest path in B and having to make repeated calls to the SP procedure (line 11). This last trick significantly improved the running time of our decoders (see Brander and Sinclair (1995) for more details and analysis).

B.2.2 Lexical Decoder Properties

One important detail is that we approximate the IBM Model 1 computation in the SSSP search by ignoring the normalizer $\mathcal{A}$ (i.e., the number of all many-to-one alignments from $\mathbf{x} \rightarrow \mathbf{z}$, shown in Equation 3.1). We do, however, use an additional data structure $l \in \mathbb{N}^{|V|}$ to store the length of the translation corresponding to the shortest path at each node from the source b (the importance of this is shown in final computation of our decoding example in Figure 3.6). Accordingly, the source node will have a length of 0, each adjacency node from the source with have a length of 0 + 1, or 1, and so on. This information can then be used for normalizing the final score when a terminating node is reached (in our case, our graphs have a



Method	$-\log p(\lambda\ \texttt{fun}_1\ \texttt{x} \mid \text{function\_1})$	$-\log p(\lambda\ \texttt{fun}_1 \mid \text{function\_1})$
Exact	1.38	1.20
Graph SSSP 1-best	1.38	?

Figure B.2: An example graph and decoding run where the lexical SSSP search does not find the correct 1-best translation (involving the excluded red edge) of the input *function_1* (uses p_t from Figure 1).

unique terminating node).

Due to this approximation, our decoder as implemented and described above is not exact, as proved by the simple counter example shown in Figure B.2 (and as already shown empirically in the chapter). In general, since the normalizer is computed at the terminating node (as opposed to during the SSSP search), longer sequences can block shorter sequences with higher (post normalized) probability. Despite this, we found this method to be empirically optimal for $k > 1$ when compared against our previous work (Richardson and Kuhn, 2017b) (as detailed in Figure 3.12). An additional implementation trick is that after each candidate SSSP is found (line 15 in Algorithm 7), we run our translation model on the input and full candidate again to compute the correct score.

C Logic Primer, Sportscaster Rule Extraction

In this section, we give a brief overview of symbolic logic notation used throughout Chapters 5-6 using examples from the air travel domain and question-answering application first discussed in Chapter 1. In Section C.2, we give a full description of the grammars used to model the Sportscaster domain in Chapter 4, using the rule set defined in Richardson and Kuhn (2016).

C.1 Knowledge Representation: How to Formally Represent Meaning

> *"The reason logic is relevant to knowledge representation and reasoning is simply that, at least according to one view, logic is the study of entailment relations – languages, truth conditions, and rules of inference. Not surprisingly, we will borrow heavily from the tools and techniques of formal symbolic logic. Specifically we will use as our first knowledge representation language a very popular logical language, that of .. first order logic (FOL) ... [which] was invented by Gottlob Frege ... for the formalization of mathematical inference, but has been co-opted by the AI community for knowledge representation purposes.. It must be stressed, however, that FOL itself is just a starting point. We will have good reasoning to consider subsets and supersets of FOL, as well as knowledge representation languages quite different in form and meaning.."*
>
> – Brachman and Levesque (2004)

The problem of knowledge representation in our setting concerns the following question: what is the nature of the output languages (shown here as $\mathcal{L}_{\texttt{sem}}$) we generate when doing NLU and SP? As discussed and motivated in Chapter 1, our output languages will often take the form of a formal logical language. In studying

$t ::= x \mid c \mid f(t_1, ..., t_n)$ *(Terms)*

$\alpha ::= P(t_1, ..., t_n) \mid \approx (t_1, t_2) \mid \bot \mid \top$ *(Atomic Formulas)*

$\alpha ::= (\neg\alpha) \mid (\alpha_1 \vee \alpha_2) \mid (\alpha_1 \rightarrow \alpha_2) \mid (\forall x.\alpha) \mid (\exists x.\alpha)$ *(Complex Formulas)*

Figure C.1: The syntax of first-order logic (FOL) formulas

such languages, we might ask: what types of formal languages do we need to define in order to capture the type of semantics we aim to model? While this question is often tied to the particular experiment or application under consideration, a reasonable starting point is the language of first-order logic (henceforth FOL), which has long been the *de facto* formal language for expressing natural language semantics in both linguistics and artificial intelligence research.

In the next section, we give a brief overview of the syntax and semantics of FOL following the review from Brachman and Levesque (2004). As pointed out in their quote above, FOL is only a starting point and most applications will require either enriching or restricting the power of the full and rather broad FOL language (for a more in-depth review, see Schubert (2015)).

C.1.1 Syntax of FOL

The FOL language consists of two types of symbols: logical symbols and non-logical symbols all defined in $\Sigma_{\texttt{sem}}$. Logical symbols include auxiliary variables, such as `(, ), ',',` the boolean connectives $\neg$*("not")*, $\wedge$*("and")*, $\vee$*("or")*, $\rightarrow$ *("if .. then")*, the existential quantifier $\exists$ (*"there exists"*) and universal quantifier $\forall$ (*"for all"*), the equality relation $\approx$, boolean symbols $\top, \bot$ (or **True**, **False**), and variables $x_1, x_2, ..., x_n \in \mathcal{X}$. In all applications of FOL, these symbols have a fixed meaning and use. In contrast, nonlogical symbols are application specific, and include predicate relation symbols $P_1, .., P_n \in \mathcal{P}$ (e.g., the relations `Flight`, `Arrive` and `Depart`), function symbols $f_1, ..., f_n \in \mathcal{F}$ (e.g., `FlightName`), and constants $c_1, c_2, ..., c_n \in \mathcal{C}$ (e.g., individual flight names such as `f105677`).

The full set of well-formed FOL formulas are those defined by the grammar

in Figure C.1, which distinguishes between *terms* and *formulas*. Function and predicate relations apply over either constant or variable arguments, where the *arity* of each relation refers to the number of arguments it takes. The `Flight` relation, for example, predicates over individual flights, such as in the formula `Flight(f105677)`, and has a single argument or an arity of one. Quantifiers apply over individual variables x in atomic formulas, such as in the following more complex formula (which we might translate into the English assertion *All flights that depart today arrive in Chicago*):

$$\forall x.\exists y.\exists z.\exists w.\texttt{Flight}(x) \wedge \texttt{Depart}(x, y, z) \wedge \texttt{Contains}(\texttt{today}, z) \rightarrow \texttt{Arrive}(x, \texttt{chicago}, w)$$

Here the universal $\forall$ and existential $\exists$ quantifiers *bind* the variables x, y, z, w. We use the predicate `Contains` to indicate that the variable z (related to a flight time) is included in the time period *today*, where *today* is represented discretely as the constant `today`. We also represent concept *Chicago* as a constant called `chicago`. The example above is an instance of a FOL *sentence*, or a FOL formula that contains no *free* or unbound variables (i.e., every argument is either a constant or bound to a quantifier). In the next section, we define a general semantics for FOL sentences.

C.1.2 Semantics and Logical Inference

FOL sentences are interpreted against models, or abstract set-theoretic descriptions of possible situations. Formally, a model is a tuple $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ consisting of a set $\mathcal{D}$ called the *domain* or *universe of discourse*, and an interpretation function $\mathcal{I}_{\mathcal{M}}$, or a mapping from nonlogical symbols in $\Sigma_{\texttt{sem}}$ to values in $\mathcal{D}$. Two example models for part of the airline domain are shown in Figure C.2, where $\mathcal{D}$ in both cases is the set $\{a, b, c, d\}$ and the constants in $\Sigma_{\texttt{sem}}$ are italicized. The general

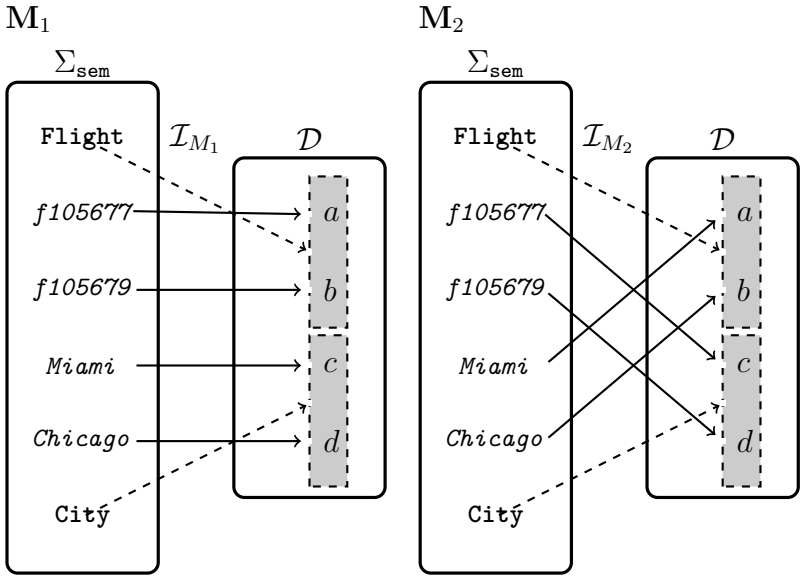


Figure C.2: An illustration of two semantic models M_1, M_2 for the airline domain.

interpretation function $\mathcal{I}$ is defined as follows:

$$\mathcal{I}(P^a) \subseteq \underbrace{\mathcal{D} \times \mathcal{D} ... \times \mathcal{D}}_{a \text{ times}} \qquad \text{(predicates } P^a \in \mathcal{P} \text{ of arity } a\text{)}$$

$$\mathcal{I}(f^a) \in \underbrace{\mathcal{D} \times \mathcal{D} ... \times \mathcal{D}}_{a \text{ times}} \to \mathcal{D} \qquad \text{(functions } f^a \in \mathcal{F} \text{ of arity } a\text{)}$$

$$\mathcal{I}(c) \in \mathcal{D} \qquad \text{(constants } c \in \mathcal{C}\text{)}$$

The symbol $\times$ is used to defined the *Cartesian power* of $\mathcal{D}$, or the set $\{(d_1, ..., d_a) \mid d_i \in \mathcal{D} \text{ for all } i = 1, ..., a\}$. The *denotation* of a given nonlogical symbol τ according to a given model $\mathcal{M}$, shown as $[\![\tau]\!]^{\mathcal{M},\mathcal{I}}$, is the interpretation of τ according to the *particular* interpretation function $\mathcal{I}_{\mathcal{M}}$ defined for that model $\mathcal{M}$. For example, the denotations of the unary predicate `Flight` in models M_1, M_2 are identical: $[\![\texttt{Flights}]\!]^{M_1,\mathcal{I}_{M_1}} = \{a, b\}$ and $[\![\texttt{Flights}]\!]^{M_2,\mathcal{I}_{M_2}} = \{a, b\}$, whereas the denotation of the constants differ markedly, for example $[\![\texttt{f105677}]\!]^{M_1,\mathcal{I}_{M_1}} = a$, $[\![\texttt{chicago}]\!]^{M_1,\mathcal{I}_{M_1}} = d$ and $[\![\texttt{f105677}]\!]^{M_2,\mathcal{I}_{M_2}} = c$, $[\![\texttt{chicago}]\!]^{M_2,\mathcal{I}_{M_2}} = b$.

Given any $\mathcal{M}$ and a *valuation function* $\mu : \mathcal{X} \to \mathcal{D}$ (i.e., a function from variables

to values in $\mathcal{D}$), we can further define the *denotation* of any arbitrary term t in a FOL sentence under a valuation μ, or $[\![t]\!]^{\mathcal{M},\mu}$, using the following recursive rules:

$$
\begin{aligned}
[\![t]\!]^{\mathcal{M},\mu} &= \mu(t) && \textbf{iff } t \in \mathcal{X} && \text{(variables)}\\
[\![t]\!]^{\mathcal{M},\mu} &= [\![t]\!]^{\mathcal{M},\mathcal{I}} && \textbf{iff } t \in \mathcal{C} && \text{(constants)}\\
[\![f(t_1,..,t_n)]\!]^{\mathcal{M},\mu} &= [\![f]\!]^{\mathcal{M},\mathcal{I}}([\![t_1]\!]^{\mathcal{M},\mu},...,[\![t_n]\!]^{\mathcal{M},\mu}) && \textbf{iff } f \in \mathcal{F} && \text{(functions)}
\end{aligned}
$$

With this background in place, it becomes possible to define the notion of *satisfiability* of a FOL sentence ϕ given a model and function μ, written as $\mathcal{M}, \mu \models \phi$ (with the negation of $\models$ written as $\not\models$):

$$
\begin{aligned}
\mathcal{M},\mu \models & P(t_1,...,t_n) && \textbf{iff } ([\![t_1]\!]^{\mathcal{M},\mu},...,[\![t_n]\!]^{\mathcal{M},\mu}) \in [\![P]\!]^{\mathcal{M},\mathcal{I}}\\
\mathcal{M},\mu \models & \approx (t_1,t_2) && \textbf{iff } [\![t_1]\!]^{M,\mu} = [\![t_2]\!]^{M,\mu}\\
\mathcal{M},\mu \models & \phi_1 \rightarrow \phi_2 && \textbf{iff } \text{ not } \mathcal{M},\mu \models \phi_1 \text{ or } \mathcal{M},\mu \models \phi_2\\
\mathcal{M},\mu \models & \phi_1 \wedge \phi_2 && \textbf{iff } \mathcal{M},\mu \models \phi_1 \text{ and } \mathcal{M},\mu \models \phi_2\\
\mathcal{M},\mu \models & \phi_1 \vee \phi_2 && \textbf{iff } \mathcal{M},\mu \models \phi_1 \text{ or } \mathcal{M},\mu \models \phi_2\\
\mathcal{M},\mu \models & \neg\phi && \textbf{iff } \mathcal{M},\mu \not\models \phi\\
\mathcal{M},\mu \models & \phi_1 \rightarrow \phi_2 && \textbf{iff } \text{ not } \mathcal{M},\mu \models \phi_1 \text{ or } \mathcal{M},\mu \models \phi_2\\
\mathcal{M},\mu \models & \exists x.\phi && \textbf{iff } \mathcal{M},\mu \models \phi \text{ for some } \mu(x) \in \mathcal{D}\\
\mathcal{M},\mu \models & \forall x.\phi && \textbf{iff } \mathcal{M},\mu \models \phi \text{ for all } \mu(x) \in \mathcal{D}
\end{aligned}
$$

Satisfiability is important because it allows us to rigorously define the conditions under which sentences are true (or false) in a model. A FOL sentence ϕ is true w.r.t. to a model $\mathcal{M}$ iff it is satisfied in that model, i.e., there exists evaluation functions $\mathcal{I}_{\mathcal{M}}, \mu$ for the particular $\mathcal{M}$ such that the above conditions are met, and false otherwise. The denotation of a FOL sentence ϕ according to a model, or $[\![\phi]\!]^{\mathcal{M}}$, is then an evaluation of the truth or falsity of the sentence. For example, $[\![\texttt{Flight(f105677)}]\!]^{M_1} = 1$ (i.e., is *true*) given that $[\![\texttt{f105677}]\!]^{M_1} = a$ and $a \in [\![\texttt{Flight}]\!]^{M_1}$, whereas the same sentence is false in M_2.

One of the main benefits of FOL, and the logical approach more generally, is that it facilitates logical inference and entailment. Formally, a FOL sentence ψ

entails another FOL sentence ϕ (or $\psi \models \phi$) if the following condition is met:

$$\psi \models \phi \textbf{ iff } \mathcal{M}, \mu \models \phi \text{ for all models } \mathcal{M}, \mu \text{ of } \psi$$

As a simple example, the sentence $\psi = \texttt{Flight}(\texttt{f105677}) \wedge \texttt{Flight}(\texttt{f105679})$ entails $\phi = \texttt{Flight}(f105679)$ since ϕ can never be false when ψ is true given the semantics of $\wedge$. More generally, entailments can also hold between sets of FOL sentences Γ and individual sentences ϕ, or $\Gamma \models \phi$.

An important property of FOL entailment is that it is *monotonic*, meaning that entailments always persist when new information is introduced. More formally, assuming a set of sentences Γ and an entailment relation $\Gamma \models \phi$, the following holds $\Gamma \cup \Gamma' \models \phi$ for any new set of sentences Γ'. It is important to note that not all inferences involving language and ordinary reasoning are monotonic, as illustrated by the following famous example:

$$\Big\{\forall x.\texttt{bird}(x) \rightarrow \texttt{fly}(x), \texttt{bird}(tweety)\Big\} \models \texttt{fly}(tweety)$$

Such an entailment does not hold when we add that `tweety` is a penguin and the rule that penguins do not fly. The difficulty is that encoding all of this information and allowing for exceptions and new information goes beyond FOL (c.f. McCarthy (1980)). In NLP, semantic inference tasks such as *recognizing textual entailment* (Dagan et al., 2005) (as looked at in Chapter 4) sometimes involve non-monotonic or *defeasible* inferences of this type, so FOL inference certainly has its limits in empirical NLP.

C.1.3 Lambda Notation

It is common in linguistics to enrich the FOL language defined above with constructs from the lambda calculus (Church, 1932). The lambda calculus is a general language and model of computation first studied in relation to natural language semantics in Montague (1970, 1973). At its core, it has an operator λ and a mechanism called *lambda abstraction*. Similar to the FOL quantifiers already introduced, λ is a kind of variable binding operator that is used to construct functions. Syntactically, lambda abstraction works in the following way: given a variable x of type

a and an expression α (e.g., a FOL formula) of type b, $\lambda x.\alpha$ returns a function (sometimes called an *anonymous* or *unammed* function) $f : a \rightarrow b$. For example:

$$\lambda x.\texttt{Flight}(x) \tag{C.1}$$

returns a function $e \rightarrow \texttt{bool}$, where we assume that x is is of type e (written as $\mathcal{D}_e$, which includes all entities in $\mathcal{D}$ already discussed), and formulas are of type $\texttt{bool}$ ($\mathcal{D}_b = \{\textbf{true}, \textbf{false}\}$), which relates to the resulting evaluation of the constituent FOL formula. In an informal sense, the lambda operator *abstracts* over and identifies the missing pieces needed to evaluate a FOL formula.

Lambda conversion replaces lambda variables with specific values, and as such performs function application. For example, the following replaces the value $\texttt{f1005677}$ with all occurrences of the the lambda variable x:

$$\lambda x.\Big[\mathit{Flight}(x)\Big](\texttt{f105677}) = \texttt{Flight}(\texttt{f105677})$$

which produces a FOL sentence that can then be evaluated accordingly.

In linguistics, the lambda calculus has one of two uses. In compositional theories of meaning, it is used as a glue language or 'special programming language' (Blackburn and Bos, 2005) for constructing complex logical sentences, as illustrated above. In SP and question-answering applications, it can also be used for describing the semantics of questions and requests. In the latter case, lambda abstraction is used to abstract over the target answers and lambda conversion is then used to evaluate candidate answers. For example, the denotation of the request *Find Flights to Chicago* can be expressed as follows using the semantics of lambda abstraction:

$$[\![\lambda x.\texttt{Flight}(x) \wedge \texttt{To}(x, \texttt{chicago})]\!]^{\mathcal{M},\mathcal{I}} = \Big\{ x \mid [\![x]\!]^{\mathcal{M},\mu} \in [\![\texttt{Flight}]\!]^{\mathcal{M},\mathcal{I}} \text{ and } ([\![x]\!]^{\mathcal{M},\mu}, [\![\texttt{chicago}]\!]^{\mathcal{M},\mathcal{I}}) \in [\![\texttt{To}]\!]^{\mathcal{M},\mathcal{I}} \Big\}$$

The lambda calculus is so ubiquitous in linguistic semantics that the famed semanticist Barbara Partee once famously remarked that "lambdas changed my life"

(Partee, 2008).

C.2 Rule Extraction in Sportscaster

C.2.1 Rule Templates

The full set of grammar rules (for both the base semantic grammars and inference grammars) used for the experiments in Chapter 4 are defined in Figure C.3 using notation from Börschinger et al. (2011)[6].

Base Semantic Grammars

The rule templates for the base semantic grammar (i.e., the grammar for generating Sportscaster `LF`s from sentences) are shown on top. As mentioned in the chapter, this grammar uses abstract role symbols, marked as variables **E** (for event predicate roles) and **A** (for argument roles), to *glue* together basic representations (according to certain word order constraints {*sv,vw..*}, see Börschinger et al. (2011) for more details). **R** and **I** then denote the associated events predicates and term predicates (respectively). Below we detail each role type:

Events: **E** includes the following roles: `player-event-intr` (which includes `defense,block,steal` event predicates), `player-event-intr2` (`kick` event predicate), `player-event-tran` (`pass`,`badPass`,`turnover`, and `block` event predicates $\in R$), `ball-event` (the `ballstopped` event predicate), and `playmode` (`corner_kick`, `kick_in`, `goal`, `kick_off`, `goal_kick`, `free_kick`,and `offside` event predicates).

Arguments: **A** includes the following: `player-arg1` and `player-arg2` (which includes all player terms `purple` 1-11 and `pink` 1-11), `game-arg1` (which includes team terms `purple-team`, `pink-team`), and `ball-arg1` (which contains a latent term `ball`). The combinations of these abstract argument types with predicates **R** and event types **E** is dictated by the original Sportscaster representation.

[6] To see the full grammars being discussed here, go to `https://github.com/yakazimir/zubr_public/blob/master/datasets/tacl_sportscaster.zip`

Each relation and term instance $\mathbf{X}$ is then associated with a *concept rule* $\mathbf{X}_c$ (as discussed in the chapter), which is then map to a set of phrase and phx rules $\mathbf{X}_p$, $\mathbf{X}_{phx}$ (as defined in Börschinger et al. (2011)). In contrast to Börschinger et al. (2011), we also use *atomic rules* that break down some of the original Sportscaster symbols into small subconcepts $\mathbf{C}$. This includes breaking down player terms into color (`purple`, `pink`) and number (`1`,..,`11`) concepts, and play events into play and team concept (e.g., `free_kick_l` to `free_kick` and `purple_team` = `l`). In these cases, the original representations can be recovered by recombining the split up pieces.

Inference Grammars

The full set of inference grammar rules are shown on the bottom of Figure C.3. Relations between semantic concepts are described here as *substition* rules, and insertions/deletions as *modifier* rules. The *function* relations enforces the single projection rule we describe in the chapter, and joins works as already described. Importantly, relations $\mathbf{S}$ are marked according to roles, and joins rules are restricted to the same role combinations as in the base semantic grammar (this is in order to avoid arbitrary joins that don't build grammatical structure in the end).

C.2.2 Alignment Computation

As described in the chapter, the inference grammar assumes as input a word/phrase alignment between sentence pairs. Such an alignment is done in a heuristic fashion by parsing each sentence individually using the base semantic grammar and aligning nodes in the resulting parse trees that have matching roles. A string is produced by pairing the yield of each matching subtree using a delimiter /. Subtrees that do not have a matching role in the other tree or are modifier expressions are isolated and aligned to the empty symbol λ. The underlying recognition algorithm for parsing was then adjusted to deal with alignment constraints.

Base Semantic Grammar Templates

word orders = $\{sv, vs, os, vo\}$

Sem_x	$\xrightarrow{glue}$	{ E A_{arg1} }	$x \in \{\text{sv,vs}\}$
Sem_x	$\xrightarrow{glue}$	{ E' (A_{arg2}) }	$x \in \{\text{ov,vo}\}$
Sem	$\xrightarrow{empty}$	$(\lambda_\text{c})\ \lambda_\text{w}$	
E	$\xrightarrow{glue}$	{ R_r (A_{arg2}) }	
E'	$\xrightarrow{glue}$	{ R_r A_{arg1} }	
R_r	$\xrightarrow{concept}$	{ R_c (λ_c) }	
A_x	$\xrightarrow{atomic}$	{ I_c (λ_c) }	$x \in \{\text{arg1,arg2}\}$
x_c	$\xrightarrow{phrase}$	{ $\text{C}_{1\text{c}}$ $\text{C}_{2\text{c}}$, ... }	$x \in \{\text{1,R}\}$
x_c	$\xrightarrow{phrase}$	x_p	$x \in \{l, R, \lambda, C\}$
x_p	$\xrightarrow{phrase}$	$(x_\text{phx})\ x_\text{w}$	
x_p	$\xrightarrow{phrase}$	$x_\text{ph}\ (\lambda_\text{w})$	
x_ph	$\xrightarrow{phrase}$	$x_\text{phx}\ (x_\text{w})$	
x_ph	$\xrightarrow{skip}$	$(\text{x}_\text{phx})\ \lambda_\text{w}$	
x_phx	$\xrightarrow{phrase}$	$(\text{x}_\text{phx})\ x_\text{w}$	
x_phx	$\xrightarrow{lexical}$	$w \in \text{corpus}$	$x \in \{l, R, \lambda, C\}$
x_w			

Inference Grammar Templates

S $\in \{\equiv, |, \#, \sqsupseteq, \sqsubseteq\}$; P $\in S \setminus \{\#, \equiv\}$,M $\in \{\sqsubseteq, \sqsupseteq, \equiv, \#\}$; $Y \in \{R, I\}$

$(\text{S} \bowtie \text{S'})_x$	$\xrightarrow{join}$	$\{\text{S}_E\ \text{S'}\ A_{arg1}\}$ $x \in \{sv, vs\}$
$(\text{S} \bowtie \text{S'})_x$	$\xrightarrow{join}$	$\{\text{S}_{E'}\ \text{S'}\ A_{arg2}\}$ $x \in \{ov, vo\}$
$\|_x$	$\xrightarrow{fun.}$	$\{\|_E\ \text{S}_A\ \}$ $x \in \{sv, vs, ...\}$
$\|_x$	$\xrightarrow{join}$	$\{\|_A\ \text{S}_E\ \}$
$(\text{S} \bowtie$	$\xrightarrow{join}$	$\{\text{S}_E\ (\text{S'}\ A_{arg2})\ \}$
$\text{S'})_E$	$\xrightarrow{join}$	$\{\text{S}_E\ \text{S'}\ A_{arg1}\ \}$
$(\text{S} \bowtie$		
$\text{S'})_{E'}$	$\xrightarrow{mod.}$	$\{\text{S}_x\ \text{M}_\text{c}\}$
$(\text{S} \bowtie \text{M})_x$	$\xrightarrow{fun.}$	$\{\text{S}_f\ \|_x\}$ $f \in \{E, A, \text{c}\}$
$\|_f$	$\xrightarrow{sub.}$	Y_c / Y'_c $x \in \{E, A\}$
P_x	$\xrightarrow{delete}$	x / λ
$\sqsubseteq_\text{c}$	$\xrightarrow{in/del}$	$\equiv_\text{c}$ / λ | λ/ $\equiv_\text{c}$
$\equiv_\text{c}$	$\xrightarrow{insert}$	λ / x
$\sqsupseteq_\text{c}$		

Examples: derivation (a), input (b) and interpretation $[\![\cdot]\!]^\mathcal{G}$ (c)

Figure C.3: The full set of rule templates for building semantic grammars for Sportscaster.

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1969). Syntax Directed Translations and the Pushdown Assembler. *Journal of Computer and System Sciences*, 3(1):37–56.

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2017). A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182*.

Allamanis, M., Tarlow, D., Gordon, A. D., and Wei, Y. (2015). Bimodal Modelling of Source Code and Natural Language. In *Proceedings of ICML*.

Allen, J. (1987). *Natural Language Understanding*. Benjamin/Cummings Publishing Company, Inc.

Allison, L. (1986). *A Practical Introduction to Denotational Semantics*. Cambridge University Press.

Andreas, J., Vlachos, A., and Clark, S. (2013). Semantic Parsing as Machine Translation. In *Proceedings of ACL*.

Angeli, G. and Manning, C. D. (2014). Naturalli: Natural Logic Inference for Common Sense Reasoning. In *Proceedings of EMNLP*.

Angeli, G., Manning, C. D., and Jurafsky, D. (2012). Parsing Time: Learning to Interpret Time Expressions. In *Proceedings of NAACL*.

Arthur, P., Neubig, G., and Nakamura, S. (2016). Incorporating Discrete Translation Lexicons into Neural Machine Translation. *Proceedings of EMNLP*.

Arthur, P., Neubig, G., Sakti, S., Toda, T., and Nakamura, S. (2015). Semantic Parsing of Ambiguous Input through Paraphrasing and Verification. *Transactions of the Association for Computational Linguistics*, 3:571–584.

Artzi, Y., Lee, K., and Zettlemoyer, L. (2015). Broad-coverage CCG Semantic Parsing with AMR. In *Proceedings of EMNLP*.

Artzi, Y. and Zettlemoyer, L. (2011). Bootstrapping Semantic Parsers from Conversations. In *Proceedings of EMNLP*.

Artzi, Y. and Zettlemoyer, L. (2013). Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Baader, F. and Nipkow, T. (1999). *Term Rewriting and All That*. Cambridge University Press.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint arXiv:1409.0473*.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Palmer, M., and Schneider, N. (2013). Abstract Meaning Representation for Sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Beesley, K. R. and Karttunen, L. (2003). Finite-state Morphology: Xerox Tools and Techniques. *CSLI, Stanford*.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 13(2):31–39.

Beltagy, I., Erk, K., and Mooney, R. (2014). Semantic Parsing using Distributional Semantics and Probabilistic Logic. *Proceedings of ACL*.

Bentivogli, L., Clark, P., Dagan, I., Dang, H., and Giampiccolo, D. (2011). The Seventh Pascal Recognizing Textual Entailment Challenge. *Proceedings of TAC*, 2011.

Berant, J., Chou, A., Frostig, R., and Liang, P. (2013). Semantic Parsing on Freebase from Question-Answer Pairs. In *Proceedings of EMNLP*.

Berant, J. and Liang, P. (2015). Imitation Learning of Agenda-based Semantic Parsers. *Transactions of the Association for Computational Linguistics*, 3:545–558.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Blackburn, P. and Bos, J. (2005). *Representation and Inference for Natural Language: A First Course in Computational Semantics*. Cambridge University Press.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461.

Bobrow, D. G. (1964). A Question-Answering System for High School Algebra Word Problems. In *Proceedings of the Fall Joint Computer Conference*.

Börschinger, B., Jones, B. K., and Johnson, M. (2011). Reducing Grounded Learning Tasks to Grammatical Inference. In *Proceedings of EMNLP*.

Bos, J. (2008). Wide-Coverage Semantic Analysis with Boxer. In *Proceedings of the 2008 Conference on Semantics in Text Processing*.

Bos, J. (2016). Expressive Power of Abstract Meaning Representations. *Computational Linguistics*, 42(3):527–535.

Brachman, R. J. and Levesque, H. J. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann.

Brander, A. and Sinclair, M. (1995). A Comparative Study of k-Shortest Path Algorithms. In *Proc. of 11th UK Performance Engineering Workshop*.

Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics*, 19(2):263–311.

Burton, R. R. (1976). Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems. *Technical Report, BBN Rep No. 3453, Bolt Beranek and Newman*.

Butt, M., Dyvik, H., King, T. H., Masuichi, H., and Rohrer, C. (2002). The Parallel Grammar Project. In *Proceedings of the 2002 Workshop on Grammar Engineering and Evaluation (GEAF)*.

Cai, Q. and Yates, A. (2013). Semantic Parsing Freebase: Towards Open-domain Semantic Parsing. *Proceedings of *SEM-2013*.

Charniak, E. (1996). *Statistical Language Learning*. MIT press.

Chen, D. L., Kim, J., and Mooney, R. J. (2010). Training a Multilingual Sportscaster: Using Perceptual Context to Learn Language. *Journal of Artificial Intelligence Research*, 37:397–435.

Chen, D. L. and Mooney, R. J. (2008a). Learning to Sportscast: A Test of Grounded Language Acquisition. In *Proceedings of ICML*, pages 128–135.

Chen, D. L. and Mooney, R. J. (2008b). Learning to Sportscast: a Test of Grounded Language Acquisition. In *Proceedings of ICML*.

Chen, D. L. and Mooney, R. J. (2011). Learning to Interpret Natural Language Navigation Instructions from Observations. In *Proceedings of AAAI*.

Cheng, J., Reddy, S., Saraswat, V., and Lapata, M. (2017). Learning Structured Natural Language Representations for Semantic Parsing. *Proceedings of ACL*.

Chiang, D. (2007). Hierarchical Phrase-based Translation. *Computational Linguistics*, 33(2):201–228.

Chiang, D., Drewes, F., Gildea, D., Lopez, A., and Satta, G. (2018). Weighted DAG Automata for Semantic Graphs. *Computational Linguistics*, 44(1):119–186.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. MIT press.

Church, A. (1932). A Set of Postulates for the Foundation of Logic. *Annals of mathematics*, pages 346–366.

Cimiano, P. and Minock, M. (2010). Natural Language Interfaces: What is the Problem?–a Data-Driven Quantitative Analysis. In *Proceedings of Natural Language Processing and Information Systems (NLDB)*.

Clark, P. (2018). What Knowledge is Needed to Solve the RTE5 Textual Entailment Challenge? *arXiv preprint arXiv:1806.03561*.

Clarke, J., Goldwasser, D., Chang, M.-W., and Roth, D. (2010). Driving Semantic Parsing from the World's Response. In *Proceedings of CONNL*.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Condoravdi, C., Richardson, K., Sikka, V., Suenbuel, A., and Waldinger, R. (2015). Natural Language Access to Data: It Takes Common Sense! In *2015 AAAI Spring Symposium Series*.

Cooper, R., Crouch, D., Van Eijck, J., Fox, C., Van Genabith, J., Jaspars, J., Kamp, H., Milward, D., Pinkal, M., and Poesio, M. (1996). Using the Framework. Technical report, Technical Report LRE 62-051 D-16, The FraCaS Consortium.

Copestake, A., Flickinger, D., Pollard, C., and Sag, I. A. (2005). Minimal Recursion Semantics: An Introduction. *Research on Language and Computation*, 3(2-3):281–332.

Copestake, A. A. and Flickinger, D. (2000). An Open Source Grammar Development Environment and Broad-coverage English Grammar Using HPSG. In *Proceedings of LREC*.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Crouch, D. and King, T. H. (2006). Semantics via F-Structure Rewriting. In *Proceedings of the LFG06 Conference*.

Crouch, R. (2005). Packed Rewriting for Mapping Semantics to KR. In *Proceedings of the 6th International Workshop on Computational Semantics*.

Dagan, I., Glickman, O., and Magnini, B. (2005). The PASCAL Recognizing Textual Entailment Challenge. In *Proceedings of the PASCAL Challenges Workshop on Recognizing Textual Entailment*.

Dahl, D. A., Bates, M., Brown, M., Fisher, W., Hunicke-Smith, K., Pallett, D., Pao, C., Rudnicky, A., and Shriberg, E. (1994). Expanding the scope of the ATIS task: The ATIS-3 corpus. In *Proceedings of the Workshop on Human Language Technology*.

Daumé III, H. (2012). A Course in Machine Learning. *Publisher, ciml*.

Davidson, D. (2001). *Essays on Actions and Events: Philosophical Essays*, volume 1. Oxford University Press on Demand.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, pages 1–38.

Deng, H. and Chrupała, G. (2014). Semantic Approaches to Software Component Retrieval with English Queries. In *Proceedings of LREC*.

Dong, L. and Lapata, M. (2016). Language to Logical Form with Neural Attention. *Proceedings of ACL*.

Dong, L. and Lapata, M. (2018). Course-to-Fine Decoding for Neural Semantic Parsing. *Proceedings of ACL*.

Dorna, M. and Emele, M. C. (1996). Semantic-based Transfer. In *Proceedings of COLING*.

Duong, L., Afshar, H., Estival, D., Pink, G., Cohen, P., and Johnson, M. (2017). Multilingual Semantic Parsing and Code-Switching. *CoNLL 2017*.

Dyer, C., Muresan, S., and Resnik, P. (2008). Generalizing Word Lattice Translation. *Proceedsings of ACL*.

Eppstein, D. (2008). K-best Enumeration. *Encyclopedia of Algorithms*, pages 1–4.

Fyodorov, Y., Winter, Y., and Francez, N. (2003). Order-based Inference in Natural Logic. *Logic Journal of IGPL*, 11(4):385–416.

Gale, W. A. and Church, K. W. (1993). A Program for Aligning Sentences in Bilingual Corpora. *Computational linguistics*, 19(1):75–102.

Gallo, G., Longo, G., Pallottino, S., and Nguyen, S. (1993). Directed Hypergraphs and Applications. *Discrete Applied Mathematics*, 42(2-3):177–201.

Gaspers, J. and Cimiano, P. (2014). Learning a Semantic Parser from Spoken Utterances. In *Proceedings of IEEE-ICASSP*.

Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in LISP*. Addison-Wesley.

Ge, R. and Mooney, R. J. (2005). A Statistical Semantic Parser that Integrates Syntax and Semantics. In *Proceedings of CONNL*.

Gildea, D. and Jurafsky, D. (2002). Automatic Labeling of Semantic Roles. *Computational Linguistics*, 28(3):245–288.

Goldberg, Y. (2016). A Primer on Neural Network Models for Natural Language Processing. *Journal of Artificial Intelligence Research*, 57:345–420.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep Learning*, volume 1. MIT press Cambridge.

Groschwitz, J., Koller, A., and Teichmann, C. (2015). Graph Parsing with S-graph Grammars. In *Proceedings of ACL*.

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016). Deep API Learning. *arXiv preprint arXiv:1605.08535*.

Haas, C. and Riezler, S. (2016). A Corpus and Semantic Parser for Multilingual Natural Language Querying of OpenStreetmap. In *Proceedings of the NAACL*.

Halevy, A., Norvig, P., and Pereira, F. (2009). The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12.

Halvorsen, P.-K. (1986). Natural Language Understanding and Montague Grammar. *Computational Intelligence*, 2(1):54–62.

Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc.

Herzig, J. and Berant, J. (2017). Neural Semantic Parsing over Multiple Knowledge-Bases. *Proceedings of ACL*.

Herzig, J. and Berant, J. (2018). Decoupling Structure and Lexicon for Zero-Shot Semantic Parsing. *arXiv preprint arXiv:1804.07918*.

Hobbs, J. R. and Rosenschein, S. J. (1977). Making Computational Sense of Montague's Intensional Logic. *Artificial Intelligence*, 9(3):287–306.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8).

Huang, L. (2008). Advanced Dynamic Programming in Semiring and Hypergraph Frameworks. In *Proceedings of COLING (tutorial notes)*.

Huang, L. and Chiang, D. (2005). Better k-best Parsing. In *Proceedings of IWPT-2005*.

Hulden, M. (2009). Foma: a Finite-State Compiler and Library. In *Proceedings of EACL*.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing Source Code using a Neural Attention Model. In *Proceedings of ACL*.

Jensen, J. L. W. V. (1906). Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta mathematica*, 30(1):175–193.

Jia, R. and Liang, P. (2016). Data Recombination for Neural Semantic Parsing. *Proceedings of ACL*.

Jie, Z. and Lu, W. (2014). Multilingual Semantic Parsing: Parsing Multiple Languages into Semantic Representations. In *Proceedings of COLING*.

Joachims, T. (2002). *Learning to Classify Text using Support Vector Machines: Methods, Theory and Algorithms*, volume 186. Kluwer Academic Publishers Norwell.

Johnson, D. B. (1977). Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13.

Johnson, M. and Goldwater, S. (2009). Improving Nonparameteric Bayesian Inference: Experiments on Unsupervised Word Segmentation with Adaptor Grammars. In *Proceedings of NAACL*.

Johnson, M., Schuster, M., Le, Q. V., Krikun, M., Wu, Y., Chen, Z., Thorat, N., Viégas, F., Wattenberg, M., Corrado, G., et al. (2016). Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *arXiv preprint arXiv:1611.04558*.

Johnson-Laird, P. N. (1977). Procedural Semantics. *Cognition*, 5(3):189–214.

Jones, B., Andreas, J., Bauer, D., Hermann, K. M., and Knight, K. (2012a). Semantics-based Machine Translation with Hyperedge Replacement Grammars. In *Proceedings of COLING*.

Jones, B. K., Johnson, M., and Goldwater, S. (2012b). Semantic Parsing with Bayesian Tree Transducers. In *Proceedings of ACL*.

Jurafsky, D., Wooters, C., Segal, J., Stolcke, A., Fosler, E., Tajchaman, G., and Morgan, N. (1995). Using a Stochastic Context-Free Grammar as a Language Model for Speech Recognition. In *Proceedings of ICASSP*.

Kamp, H. and Reyle, U. (2013). *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*, volume 42. Springer Science & Business Media.

Kasami, T. (1965). An Efficient Recognition and Syntax Algorithm for Context-Free Languages. *Technical Report, Air Force Cambridge Research Lab.*

Kate, R. J., Wong, Y. W., and Mooney, R. J. (2005). Learning to Transform Natural to Formal Languages. In *Proceedings of the National Conference on Artificial Intelligence*.

Kim, J. and Mooney, R. J. (2010). Generative Alignment and Semantic Parsing for Learning from Ambiguous Supervision. In *Proceedings of COLING*, pages 543–551.

Kim, J. and Mooney, R. J. (2012). Unsupervised PCFG Induction for Grounded Language Learning with Highly Ambiguous Supervision. In *Proceedings of EMNLP-CoNLL*.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). Robocup: The Robot World Cup Initiative. In *Proceedings of the First International Conference on Autonomous Agents*.

Klein, D. and Manning, C. (2004). Parsing and Hypergraphs. In *New Developments in Parsing Technology*, pages 351–372. Springer.

Knight, K. (1999). Decoding Complexity in Word-Replacement Translation Models. *Computational Linguistics*, 25(4):607–615.

Knight, K. and Al-Onaizan, Y. (1998). Translation with Finite-state Devices. In *Proceedings of AMTA*.

Knuth, D. (1977). A Generalization of Dijkstra's Algorithm. *Information Processing Letters*, 6(1):1—5.

Knuth, D. E. (1968). Semantics of Context-Free Languages. *Mathematical systems theory*, 2(2):127–145.

Kočiský, T., Melis, G., Grefenstette, E., Dyer, C., Ling, W., Blunsom, P., and Hermann, K. M. (2016). Semantic Parsing with Semi-Supervised Sequential Autoencoders. In *Proceedings of EMNLP*.

Koehn, P. (2005). Europarl: A Parallel Corpus for Statistical Machine Translation. In *Proceedings of MT summit*.

Koehn, P. (2009). *Statistical Machine Translation*. Cambridge University Press.

Koehn, P., Och, F. J., and Marcu, D. (2003). Statistical Phrase-based Translation. In *Proceedings of NAACL*.

Koller, A. (2015). Semantic Construction with Graph Grammars. In *Proceedings of IWCS*, pages 228–238.

Krishnamurthy, J., Dasigi, P., and Gardner, M. (2017). Neural Semantic Parsing with Type Constraints for Semi-structured Tables. In *Proceedings of EMNLP*.

Krishnamurthy, J. and Mitchell, T. M. (2012). Weakly Supervised Training of Semantic Parsers. In *Proceedings of EMNLP*.

Kushman, N., Artzi, Y., Zettlemoyer, L., and Barzilay, R. (2014). Learning to Automatically Solve Algebra Word Problems. In *Proceedings of ACL*.

Kushman, N. and Barzilay, R. (2013). Using Semantic Unification to Generate Regular Expressions from Natural Language. In *Proceedings of NAACL*.

Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., and Steedman, M. (2010). Inducing Probabilistic CCG Grammars from Logical form with Higher-Order Unification. In *Proceedings of EMNLP*.

Lafferty, J. D. (2000). *A Derivation of the Inside-Outside Algorithm from the EM Algorithm*. IBM TJ Watson Research Center.

Lari, K. and Young, S. (1990). The Estimation of Stochastic Context-free Grammars using the Inside-Outside Algorithm. *Computer Speech and Language*, 4(1):35–56.

Lawler, E. L. (1972). A Procedure for Computing the k Best Solutions to Discrete Optimization Problems and its Application to the Shortest Path Problem. *Management science*, 18(7):401–405.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lewis II, P. M. and Stearns, R. E. (1968). Syntax-Directed Transduction. *Journal of the ACM*, 15(3):465–488.

Li, J., Zhu, M., Lu, W., and Zhou, G. (2015). Improving Semantic Parsing with Enriched Synchronous Context-Free Grammar. In *EMNLP*, pages 1455–1465.

Li, P., Liu, Y., and Sun, M. (2013). An Extended GHKM Algorithm for Inducing Lambda-SCFG. In *AAAI*.

Liang, P. (2013). Lambda Dependency-Based Compositional Semantics. *arXiv preprint arXiv:1309.4408*.

Liang, P. (2016). Learning Executable Semantic Parsers for Natural Language Understanding. *Communications of the ACM*, 59(9):68–76.

Liang, P., Jordan, M. I., and Klein, D. (2011). Learning Dependency-Based Compositional Semantics. In *Proceedings of ACL*.

Liang, P., Jordan, M. I., and Klein, D. (2013). Learning Dependency-based Compositional Semantics. *Computational Linguistics*, 39(2):389–446.

Liang, P. and Potts, C. (2015). Bringing Machine Learning and Compositional Semantics Together. *Annual Review of Linguistics*, 1(1):355–376.

LoBue, P. and Yates, A. (2011). Types of Common-sense Knowledge Needed for Recognizing Textual Entailment. In *Proceedings of ACL-HLT*.

Lopez, A. (2008). Statistical Machine Translation. *ACM Computing Surveys*, 40(3):8.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective Approaches to Attention-Based Neural Machine Translation. *arXiv preprint arXiv:1508.04025*.

Lv, F., Zhang, H., Lou, J.-g., Wang, S., Zhang, D., and Zhao, J. (2015). Codehow: Effective Code Search based on API Understanding and Extended Boolean Model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE.

MacCartney, B. (2009). *Natural Language Inference*. PhD thesis, Department of Computer Science,Stanford University.

MacCartney, B. and Manning, C. (2008). Modeling Semantic Containment and Exclusion in Natural Language Inference. In *Proceedings of COLING*.

MacCartney, B. and Manning, C. D. (2009). An Extended Model of Natural Logic. In *Proceedings of IWCS*.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT press.

Manshadi, M. H., Gildea, D., and Allen, J. F. (2013). Integrating Programming by Example and Natural Language Programming. In *AAAI*.

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195.

McCarthy, J. (1980). Circumscription—a Form of Non-monotonic Reasoning. *Artificial intelligence*, 13(1):27–39.

Melamed, I. D. (2004). Statistical Machine Translation by Parsing. In *Proceedings of ACL*.

Miceli Barone, A. V. and Sennrich, R. (2017). A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. *arXiv preprint arXiv:1707.02275*.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT press.

Misra, D. K. and Artzi, Y. (2016). Neural Shift-Reduce CCG Semantic Parsing. In *EMNLP*.

Mohri, M. (1996). On Some Applications of Finite-State Automata Theory to Natural Language Processing. *Natural Language Engineering*, 2(1):61–80.

Montague, R. (1970). Universal Grammar. *Theoria*, 36(3):373–398.

Montague, R. (1973). The Proper Treatment of Quantification in Ordinary English. In *Philosophy, Language, and Artificial Intelligence*, pages 141–162. Springer.

Mooney, R. (2007a). Learning for Semantic Parsing. In *Proceedings of CICLing*.

Mooney, R. (2007b). Learning for Semantic Parsing. In *Proceedings of CICLing*.

Mooney, R. (2008). Learning to Connect Language and Perception. In *Proceedings of AAAI*.

Neal, R. M. and Hinton, G. E. (1998). A View of the EM Algorithm that Justifies Incremental, Sparse, and other Variants. In *Learning in Graphical Models*, pages 355–368. Springer.

Nederhof, M.-j. and Satta, G. (2010). Theory of Parsing. *The Handbook of Computational Linguistics and Natural Language Processing*, pages 105–130.

Neubig, G. (2017). Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial. *arXiv preprint arXiv:1703.01619*.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. (2017). Dynet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980*.

Nielsen, L. R., Andersen, K. A., and Pretolani, D. (2005). Finding the k shortest hyperpaths. *Computers & Operations Research*, 32(6):1477–1497.

Och, F. J. and Ney, H. (2003). A Systematic Comparison of Various Statistical Alignment Models. *Computational Linguistics*, 29(1):19–51.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE.

Östling, R. and Tiedemann, J. (2017). Neural Machine Translation for Low-resource Languages. *arXiv preprint arXiv:1708.05729*.

Parsons, T. (1990). *Events in the Semantics of English*, volume 5. Cambridge, MA: MIT Press.

Partee, B. (2005). Montague, Richard (1930-1981). In Brown, K., editor, *Encyclopedia of Language and Linguistics*. Oxford: Elsevier.

Partee, B. (2008). *Compositionality in Formal Semantics: Selected Papers by Barbara Partee*. John Wiley & Sons.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of ICML*.

Pasupat, P. and Liang, P. (2015). Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of ACL*.

Pavlick, E., Bos, J., Nissim, M., Beller, C., Van Durme, B., and Callison-Burch, C. (2015). Adding Semantics to Data-Driven Paraphrasing. In *Proceedings of ACL*.

Peng, X., Song, L., and Gildea, D. (2015). A Synchronous Hyperedge Replacement Grammar based approach for AMR Parsing. *CoNLL 2015*.

Peng, X., Wang, C., Gildea, D., and Xue, N. (2017). Addressing the Data Sparsity Issue in Neural AMR Parsing. *Proceedings of ACL*.

Popescu, A.-M., Etzioni, O., and Kautz, H. (2003). Towards a Theory of Natural Language Interfaces to Databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI)*.

Quernheim, D. and Knight, K. (2012). Towards Probabilistic Acceptors and Transducers for Feature Structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*.

Quirk, C., Mooney, R. J., and Galley, M. (2015). Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proceedings of ACL*.

Rabinovich, M., Stern, M., and Klein, D. (2017). Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of ACL*.

Rattan, D., Bhatia, R., and Singh, M. (2013). Software Clone Detection: A Systematic Review. *Information and Software Technology*, 55(7):1165–1199.

Reddy, S., Lapata, M., and Steedman, M. (2014). Large-scale Semantic Parsing without Question-Answer Pairs. *Transactions of the Association of Computational Linguistics*, 2(1):377–392.

Richardson, K. (2017). Code-Datasets. `https://github.com/yakazimir/Code-Datasets`.

Richardson, K. (2018). A Language for Function Signature Representations. *arXiv preprint arXiv:1804.00987*.

Richardson, K., Berant, J., and Kuhn, J. (2018). Polyglot Semantic Parsing in APIs. In *Proceedings of NAACL*.

Richardson, K. and Kuhn, J. (2012). Light Textual Inference for Semantic Parsing. *Proceedings of COLING*.

Richardson, K. and Kuhn, J. (2014). Unixman Corpus: A Resource for Language Learning in the Unix Domain. In *Proceedings of LREC*.

Richardson, K. and Kuhn, J. (2016). Learning to Make Inferences in a Semantic Parsing Task. *Transactions of the Association of Computational Linguistics*, 4(1):155–168.

Richardson, K. and Kuhn, J. (2017a). Function Assistant: A Tool for NL Querying of APIs. In *Proceedings of EMNLP*.

Richardson, K. and Kuhn, J. (2017b). Learning Semantic Correspondences in Technical Documentation. In *Proceedings of ACL*.

Richardson, K., Zarrieß, S., and Kuhn, J. (2017). The Code2Text Challenge: Text Generation in Source Code Libraries. In *Proceedings of INLG*.

Riezler, S. (2014). On the Problem of Theoretical Terms in Empirical Computational Linguistics. *Computational Linguistics*, 40(1):235–245.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological review*, 65(6):386.

Rounds, W. C. (1970). Mappings and Grammars on Trees. *Theory of Computing Systems*, 4(3):257–287.

Russell, B. (1995). *My Philosophical Development*. Psychology Press.

Schubert, L. (2015). Semantic Representation. In *Proceedings of AAAI*.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural Machine Translation of Rare Words with Subword Units. *arXiv preprint arXiv:1508.07909*.

Shieber, S. M. and Schabes, Y. (1990). Synchronous Tree-Adjoining Grammars. In *Proceedings of COLING*.

Simmons, R. F. (1965). Answering English Questions by Computer: A Survey. *Communications of the ACM*, 8(1):53—70.

Smith, N. A. (2011). Linguistic Structure Prediction. *Synthesis Lectures on Human Language Technologies*, 4(2):1–274.

Snow, R., O'Connor, B., Jurafsky, D., and Ng, A. Y. (2008). Cheap and fast—but is it good?: Evaluating Non-expert Annotations for Natural Language Tasks. In *Proceedings of EMNLP*.

Steedman, M. (1996). *Surface Structure and Interpretation*. MIT press.

Susanto, R. H. and Lu, W. (2017). Semantic Parsing with Neural Hybrid Trees. In *AAAI*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In *Proceedings of NIPS*.

Tang, L. R. and Mooney, R. J. (2000). Automated Construction of Database Interfaces: Integrating Statistical and Relational Learning for Semantic Parsing. In *Proceedings of SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*.

Tian, R., Miyao, Y., and Matsuzaki, T. (2014). Logical Inference on Dependency-based Compositional Semantics. *Proceedings of ACL*.

Toledo, A., Alexandropoulou, S., Katrenko, S., Klockmann, H., Kokke, P., and Winter, Y. (2013). Semantic Annotation of Textual Entailment. In *Proceedings of IWCS*.

Tsvetkov, Y., Sitaram, S., Faruqui, M., Lample, G., Littell, P., Mortensen, D., Black, A. W., Levin, L., and Dyer, C. (2016). Polyglot Neural Language Models: A Case Study in Cross-Lingual Phonetic Representation Learning. *arXiv preprint arXiv:1605.03832*.

Turney, P. D. and Pantel, P. (2010). From Frequency to Meaning: Vector Space Models of Semantics. *Journal of Artificial Intelligence Research*, 37:141–188.

Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.-C., Gerber, D., and Cimiano, P. (2012). Template-based Question Answering over RDF Data. In *Proceedings of the 21st international conference on World Wide Web*.

van Benthem, J. (1986). *Essays in Logical Semantics*. Springer.

Van Eijck, J. and Unger, C. (2010). *Computational Semantics with Functional Programming*. Cambridge University Press.

Waldinger, R. J., Bobrow, D. G., Condoravdi, C., Richardson, K., and Das, A. (2011). Accessing Structured Health Information through English Queries and Automatic Deduction. In *AAAI Spring Symposium: AI and Health Communication*.

Wang, A., Kwiatkowski, T., and Zettlemoyer, L. (2014). Morpho-syntactic Lexical Generalization for CCG Semantic Parsing. In *Proceedings of EMNLP*.

Werbos, P. J. (1990). Backpropagation Through Time: What it Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560.

Widdows, D. (2004). *Geometry and Meaning*. CSLI Publications.

Williams, P., Sennrich, R., Post, M., and Koehn, P. (2016). Syntax-based Statistical Machine Translation. *Synthesis Lectures on Human Language Technologies*, 9(4):1–208.

Wong, Y. W. and Mooney, R. J. (2006). Learning for Semantic Parsing with Statistical Machine Translation. In *Proceedings of NAACL*.

Wong, Y. W. and Mooney, R. J. (2007). Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus. In *Proceedings of ACL*.

Woods, W. A. (1968). Procedural Semantics for a Question-answering Machine. In *Proceedings of the Fall Joint Computer Conference (AFIPS)*.

Woods, W. A. (1970). Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, 13(10):591–606.

Woods, W. A. (1973). Progress in Natural Language Understanding: An Application to Lunar Geology. In *Proceedings of the National Computer Conference and Exposition*.

Woods, W. A. (1978). Semantics and Quantification in Natural Language Question Answering. *Advances in Computers*, 17:1–87.

Wu, D. (1997). Stochastic Inversion Transduction Grammars and Bilingual Parsing of Parallel Corpora. *Computational Linguistics*, 23(3):377–403.

Yen, J. Y. (1971). Finding the k Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716.

Yin, P. and Neubig, G. (2017). A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of ACL*.

Zarrieß, S. and Richardson, K. (2013). An Automatic Method for Building a Data-to-Text Generator. In *Proceedings of ENLG*.

Zelle, J. M. and Mooney, R. J. (1996). Learning to Parse Database Queries Using Inductive Logic Programming. In *Proceedings of AAAI*.

Zettlemoyer, L. S. and Collins, M. (2009). Learning Context-Dependent Mappings from Sentences to Logical Form. In *Proceedings of ACL*.

Zettlemoyer, L. S. and Collins, M. (2012). Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. *arXiv preprint arXiv:1207.1420*.

Zhai, C. and Lafferty, J. (2004). A Study of Smoothing Methods for Language Models Applied to Information Retrieval. *ACM Transactions on Information Systems*, 22(2):179–214.

Zhang, Y., Pasupat, P., and Liang, P. (2017). Macro Grammars and Holistic Triggering for Efficient Semantic Parsing. In *Proceedings of EMNLP*.

Zollmann, A. and Venugopal, A. (2006). Syntax Augmented Machine Translation via Chart Parsing. In *Proceedings of the Workshop on Statistical Machine Translation*.

Zoph, B., Yuret, D., May, J., and Knight, K. (2016). Transfer Learning for Low-resource Neural Machine Translation. *Proceedings of ACL*.