Translating Natural Language into Source Code Via Tree Transduction

Rafael - Michael Karampatsis

Master of Science by Research
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2015
Abstract

Semantic parsing is the task of transforming a natural language sentence into a computer executable complete meaning representation. It is also the basis of natural language understanding, which is one of the oldest and most studied problems in the field of artificial intelligence. This thesis describes a semantic parser that translates a natural language sentence into source code based on a tree-to-tree transducer. The core of the method is the extraction of a synchronous tree substitution grammar that translates an elementary tree in the source language into in the target language. Although the method focuses on translating a natural language specification into source code, it can be used to perform the opposite task (generating language from source code) as well. Moreover, the very same methodology can be applied to perform syntax based machine translation from one language to another but it requires a parser for both languages. Lastly, the method is compared and evaluated against two baselines on two different datasets, one based on Excel commands, and one based on queries for a geography database.

The main contributions are that this is the first method to perform tree-to-tree translation, as previous methods were based on tree-to-string translation, it is based on a generative model, whilst previous approaches employ a generative model, and lastly, it is the first statistical method to perform experiments on the Excel dataset. Finally, ideas for extensions or modifications that could improved the method are proposed.
Acknowledgements

I would like to thank my supervisors, Professor Mirella Lapata and Dr. Charles Sutton, for their consistent and restless guidance, advice, discussions, support, excellent collaboration and patience during the creation of this thesis. I would also like to thank my fellow students in the CDT in Data science for this wonderful year. Special thanks, to my family for their support, love, and encouragement to never quit on my dreams, even while leaving abroad. Finally, last but not least, I would also like to thank all the people who stood by me during this process.

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Rafael - Michael Karampatsis)
Table of Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................... 1
   1.2 Outline ........................................... 3

2 Background ........................................... 5
   2.1 Related Work ........................................ 5
   2.2 Programming Language ......................... 8
       2.2.1 Abstract Syntax Tree .................. 8
   2.3 Grammars ........................................ 9
       2.3.1 Grammar Derivation ....................... 9
       2.3.2 Regular Grammar ......................... 10
       2.3.3 Context Free Grammar ................. 10
       2.3.4 BNF ....................................... 11
       2.3.5 Regular Tree Grammar ................ 11
       2.3.6 Synchronous Grammars ............... 12
   2.4 Tree Transducers ................................ 13

3 Machine Translation as Tree Transduction ........... 17
   3.1 Word Alignments Extraction .................. 18
   3.2 Grammar Extraction ............................... 19
       3.2.1 Translation Rules ....................... 20
       3.2.2 Copy Rules ................................ 22
       3.2.3 Epsilon Rules ............................. 23
   3.3 Language Model .................................... 23
   3.4 Feature Extraction ............................... 23
       3.4.1 T3 Features ................................ 23
       3.4.2 New Features ............................... 24
List of Figures

2.1 An example AST for the expression: 3 * 2 + (6 + 5 + 2 * 8). ........ 9
2.2 An example grammar and its possible derivations. .................. 14
2.3 A simple BNF grammar example. ................................. 14
2.4 Example of an RTG grammar and the language recognized by it. .. 15
2.5 Example of an SCFG, its derivation steps, and the full derivation. .. 16

3.1 Example of a predicted many-to-many word alignment for an instance of the Geoquery dataset. Source sentence: How many people live in StateName_new_mexico ? Target sentence: answer population_1 stateid new_mexico .................. 20
3.2 Constituent alignment example from Cohn and Lapata (2009). .......... 21

4.1 Example task: Total of amountdue for all invoices with desc of lamp. . 30
4.2 Application example of the normalization algorithm on one of the Excel dataset instances. .................................................. 35
4.3 Application example of the normalization algorithm on one of the Excel dataset instances. .................................................. 35
4.4 Merging of NNP nodes that are parts of the same named entity. ....... 37
4.5 Creation of an AST from code tokens. Code tokens are shown in blue color. .............................................................. 39
4.6 Creation of an AST from code tokens. Code tokens are shown in blue color. .............................................................. 39

5.1 Conversion of a tree into a flat sequence of tokens using a pre-order left-to-right traversal (tree linearisation). ............................... 42

6.1 Example output for the various methods for the sentence “What are the highest points of all the states”. ................................. 53
List of Tables

4.1 Geoquery instance example: Natural language question in English and its equivalent Prolog and functional queries. ....................... 36
4.2 Analogous predicates and their POS tag. .......................... 38

6.1 Experimental results for the character edit distance baseline on Excel. 47
6.2 Experimental results for the character edit distance baseline on Geoquery. 48
6.3 Experimental results for the phrase-based MT baseline on the Excel dataset. ......................................................... 49
6.4 Experimental results for the phrase-based MT baseline on the Geoquery dataset. ..................................................... 50
6.5 Experimental results for the tree transducer method on the Excel dataset. 51
6.6 Experimental results for the tree transducer method on the Geoquery dataset. ......................................................... 52
6.7 Comparison between the different methods on the Excel dataset. . . 52
6.8 Comparison between the different methods in the Geoquery dataset. . 53
Chapter 1

Introduction

1.1 Motivation

Natural language understanding (NLU) is one of the oldest and most studied problems in the field of artificial intelligence (AI). The basis of NLU is the task of parsing and translating some input in natural language into a well-defined formal meaning representation that a computer can understand and use. For example, this meaning representation could be in first-order logic. This task is commonly known as the task of semantic parsing and has been studied extensively (Zelle, 1995; Zelle and Mooney, 1996; Papineni et al., 1997; Ge and Mooney, 2005; Zettlemoyer and Collins, 2005; Wong and Mooney, 2006; Kate and Mooney, 2006; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Lu et al., 2008; Ge and Mooney, 2009; Jones et al., 2012; Andreas et al., 2013; Quirk et al., 2015). However, although there has been extensive work, semantic parsing is still an interesting open problem.

In this thesis, we study a methodology for semantic parsing of a sentence in natural language into some form of executable source code. The problem is cast as syntax-based machine translation by employing a tree transducer that learns tree-to-tree mappings from a natural language parse to an abstract syntax tree of code. The method extracts a grammar of translation rules, which are used by the tree transducer to create possible derivations that translate the input parse tree into an abstract syntax tree of code in the meaning representation language. These derivations are scored using a discriminative model of features for the rules of which the derivation consists of, as well as the score of a bigram language model for the target side. Lastly, the best scoring derivation is approximated and returned, by using a chart-based dynamic programming algorithm, coupled with beam-search.
The tree-to-tree transducer ensures that the generated semantic parses will be syntactically correct, which other methods such as phrased-based machine translation do not. This is a very important element of the problem that has to addressed from any method that aims to produce an executable syntactic parse, since code lacking proper syntax can neither be run nor produce the proper result. Also, unlike other methods it does not need any hand-built resources such as hand-crafted rules. Additionally, this methodology allows the system to also perform the reverse task (natural language generation from a semantic parse), something that is not possible by other methods (e.g., CCG grammars). Moreover, the proposed methodology does not require strict isomorphism between the natural language and the meaning representation. In addition, the proposed methodology can easily be applied to perform syntax-based machine translation between two natural languages such as English and French (with the small requirement of a parser for both languages). Hence, this method offers a general framework that can easily be used for many different tasks as it can be used to perform monolingual translation, machine translation, semantic parsing, and language generation.

Experiments on two datasets were performed in order to evaluate the method. The first is a very common dataset in the semantic parsing literature called Geoquery (Zelle, 1995), which contains queries expressed in natural language for a database of American geography along with their meaning representation (query) that can be executed by the database. While the second consists of Excel commands and their description in natural language (Gulwani and Marron, 2014).

The motivation for this work comes from recent attempts to translate natural language into source code and vice versa. For example Wong et al. (2013); Movshovitz-Attias and W. Cohen (2013); Wong et al. (2015) attempt to automatically create comments from source code or create chunks of code from a natural language description (Gulwani and Marron, 2014) by employing parallel corpus of code and its description. In addition, Karaivanov et al. (2014) built a system that translates C# code to Java by using a parallel corpus of 20499 C#-to-Java method translations. These attempts have motivated us to create a system that translates natural language into code and code into natural language. This system could be used to help inexperience users to create code by allowing them to write the description in natural language. Also, it could be used to simplify the process of documenting the code and allow other programmers to understand it quickly by generating an explanation of the code in natural language.

This project mostly focuses on the first direction (translating natural language into
code) as the reverse direction is a natural language generation task and should be evaluated as one by using appropriate metrics. One of the main reasons for this is that the same meaning representation corresponds to multiple different sentences with the same meaning in natural language. However, the method could be easily be applied on the reverse direction (which could have a major impact to the software engineering process), by applying the following two simple modifications to the method:

1. A ranked list of the top k derivations instead of the best scoring derivation is returned.

2. The evaluation is performed by using an appropriate metric for natural language generation or by humans.

1.2 Outline

The rest of this document is organized as follows: In Chapter 2, the background related to this project is provided and then core natural language processing concepts are briefly introduced. Chapter 3 introduces a novel method for translating natural language into source code and vice versa based on a tree transducer. Chapter 4 describes the datasets that were used during this project. In Chapter 5, two baselines that were implemented in order to compare the method against are described. Chapter 6 describes the experiments conducted, the evaluation metrics used, and finally discusses and analyses the results of the baselines, and the proposed method. Finally, in Chapter 7 the main contributions and a summary of this project are discussed, along with future work that aims to extend and improve the method.
Chapter 2

Background

2.1 Related Work

Related work can be organized in two main branches. The first branch is composed of works that aim to facilitate and speed up software development. The effort focuses on attempts, which either seek to automate code commenting or to simplify the process of code creation. On the contrary, the second branch consists of works, which have focused on grounding natural language into a formal meaning representation. Most works that address this problem refer to this task as semantic parsing.

Software development has been one of the most major businesses for the last two decades. Every application that we use in our computers, tablets, and smartphones has been built by software developers. However, understanding and maintaining these vast amounts of code can be really hard even for experienced software developers. To facilitate this process developers rely on code comments written in natural language to help them understand code easier. Automation of the code commenting process could save a significant amount of time for developers. This has influenced a lot of people to create techniques that generate comments for Java methods from natural language by directly using the code elements. These techniques however might fail to generate accurate comments if the source code contains poorly named identifiers or method names.

Wong et al. (2013) mined Stack Overflow, a popular question and answer website, in order to extract code segments and their corresponding human written descriptions for automatic comment generation. This approach however cannot generate comments for code segments that are not similar to those that exist in the question and answer website. To overcome this problem in a follow up work Wong et al. (2015) mined
open source software repositories. They downloaded 1,005 Java open source projects from GitHub that contain 42 million lines of code and 17 millions lines of comments. Their approach identifies similar code segments between two code repositories. They based their method on the idea that software reuse is common and millions of lines of open source projects that contain code comments are available for the generation of human written comments.

On the other hand, there has been a lot of work which has focused on simplifying the process of creating code. Several approaches have focused on keyword programming. It is a technique that allows the user to provide a set of unordered keywords and translates them into valid source code. As a result it effectively reduces the need to remember details of programming language syntax and APIs. The keywords are utilized as a query that searches the space of valid expressions. When keyword programming is employed, it is very common to return a set of ranked programs instead of just one. Little and Miller (2007) showed that when a highly type constrained language like Java is being generated then a small number of keywords is often sufficient to generate the correct method calls. Some approaches seek to translate a richer natural language description directly into code. For example Gulwani and Marron (2014) implemented NLyze. NLyze is an Excel add-on which converts a natural language specification into an Excel script\(^1\) for an Excel spreadsheet which they take as context. Their method combines ideas both from keyword programming and semantic parsing and also produces a set of ranked likely programs.

Semantic parsing is the task of transforming a natural language sentence into a computer executable complete meaning representation (MR), which is called a semantic parse, for a domain-specific application. As a consequence, the task is domain and application dependent. Semantic parsing is very useful as it offers an interface for some computing application. For instance, one can ask a database query in natural language (Zelle and Mooney, 1996). Since a semantic parse is a formal language aimed to be run by a computer, it is crucial to get the exact MR, because otherwise the computer will generate the wrong output (there is no room for mistakes).

A semantic parse is similar to a syntactic parse but it also includes semantics. Moreover, it can be considered as another natural language (Papineni et al., 1997; Wong and Mooney, 2006). Translating natural language into a semantic parse as well as doing the opposite can both be considered as performing machine translation (Pap-

\(^1\)The programs are generated in a Domain Specific Language (DSL) which is equivalent to Excel commands.
2.1. Related Work

ineni et al., 1997). Finally, the reverse direction (i.e., translating a semantic parse into natural language) can be considered as the task of natural language generation (NLG) (Wong and Mooney, 2007).

There have been various approaches for semantic parsing. Zettlemoyer and Collins (2005, 2007) learn a probabilistic semantic parsing model which initially requires a hand-built, ambiguous CCG grammar (Steedman, 1987) template. Wong and Mooney (2006, 2007) cast the problem as syntax-based machine translation using an SCFG to model the translation of natural language into a formal MR language. While Kate and Mooney (2006) use an SVM with string kernels to learn a robust semantic parser that labels substrings of the natural language with entities from the MR. Lu et al. (2008) use a hybrid tree semantic parse based on a generative model, which simultaneously generates a natural language sentence and an MR structure. Following this idea, Jones et al. (2012) use a tree-to-string transducer coupled with a generative model. Ge and Mooney (2005, 2009) exploit an existing syntactic parser to produce disambiguated parse trees that drive the compositional semantic interpretation process. Lastly, Andreas et al. (2013) approach the problem as a straightforward machine translation task and they experiment with both a phrase-based and an hierarchical translation model.

This work’s objective is similar to that of (Quirk et al., 2015), as it is mostly an attempt to join the two branches. Quirk et al. (2015) present an approach that learns to map natural-language descriptions of simple “if-then” rules to executable code. Their objective is to build a semantic parser that allows users to describe code “recipes” (“if-then” rules) in natural language and have them automatically mapped to executable code. These recipes were collected from the IFTTT website. These “if-then” rules allow users to control many aspects of their digital life such as smart devices. Their method is based on choosing the semantic derivation \( D \) with best probability given a sentence \( E \). The probability of a semantic derivation is defined as:

\[
P(D|E) = \prod_{(r, \ldots, j) \in D} P(r, i \ldots j|E)
\]

Where \( i \) and \( j \) are positions in the sentence \( E \) and \( i < j \). The model assumes that each production is independent of all others, and that it is conditioned only on the string to which it is aligned. To measure the probability of a production they use logistic regression classifiers with word unigram, word bigram, and character trigram features.

The methodology employed in this work bares many similarities with the approaches of Wong and Mooney (2006, 2007) and the tree transducer based method of Jones et al.

\[^2\text{http://ifttt.com}\]
However, it has two major differences from them. Firstly, it utilizes a tree-to-
tree transducer that translates an input parse tree of sentence in natural language into
an abstract syntax tree of code, which is the meaning representation of the sentence.
Secondly, it does not use a generative model, but instead it uses a discriminative model
to approximate the best possible derivations that translates an input tree of the source
language into a tree of the target language. Additionally, the method can also perform
the reverse task (i.e., generate language from source code) without any changes. This
problem is not explored in this work, as it would be more suitable for it to be evalu-
ated as an NLG task using appropriate metrics (e.g., BLEU (Papineni et al., 2002)
or ROUGE-SU4 (Lin, 2004) ) or human evaluation. The reason for this is that, while
there is only one correct meaning representation, there are multiple sentences in natural
language that have the same meaning.

2.2 Programming Language

A programming language is a syntactic realization of one or more computational mod-
els. A complete description of a programming language includes the computational
model, the syntax and semantics of programs, and the pragmatic considerations that
shape the language. The relationship between the syntax and the computational model
is provided by a semantic description. Semantics provide meaning to programs. The
computational model provides much of the intuition behind the construction of pro-
grams. When a programming language is faithful to the computational model, pro-
grams can be more easily written and understood (Aaby, 1996). The rest of this section
focuses on the syntax of programs.

2.2.1 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree that represents the syntactic structure of source
code in a programming language. As the name suggests the syntax is abstract and does
not represent every detail of the real syntax. For example brackets or parentheses, or
a semicolon at the end of a statement can be skipped. This is in contrast to a parse
tree (see Section 2.3.1), where the syntax is concretely described. Figure 2.1 shows an
example of an AST for the expression: $3 \times 2 + (6 + 5 + 2 \times 8)$. 
2.3. Grammars

A grammar is defined as 4-tuple $G = (V, \Sigma, R, S)$.

$V$: A finite alphabet of symbols used in the grammar.

$\Sigma \subseteq V$: A finite set of terminal symbols, which are the elementary symbols of the language defined by the grammar and cannot be changed by a rule of the grammar.

$N = V - \Sigma$: A finite set of non-terminal symbols, which can be replaced by groups of non-terminal or terminal symbols.

$R \subseteq (V^+ \times V^*)$: A finite set of productions rules (usually referred as productions) e.g., $NNP \rightarrow new\_mexico$.

$S \in V - \Sigma$: A special non-terminal symbol also called a start symbol, which appears in the initial string generated by the grammar.

The following notation will be used for the rest of this chapter. Non-terminal symbols (elements of $V - \Sigma$) will be starting with an upper-case letter (e.g., $NN$ or $State\_idP$), terminal symbols (elements of $\Sigma$) will be mentioned using lower-case letters (e.g., $city$ or $state\_id$) and $\epsilon$ represents the empty string (a string of length 0).

### 2.3.1 Grammar Derivation

A derivation of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs in the language defined by the grammar. To fully specify a derivation the exact rule and the non-terminal on which it is applied must be specified in each step of the derivation. Alternatively, a derivation can be fully specified using a tree, which is also called a parse tree (Chiswell and Hodges, 2007) or derivation tree or (concrete) syntax tree. The
main difference from an AST is that the structure of the tree reflects more concretely
the syntax of the input language.

An example grammar coupled with its four possible derivations are illustrated il-
illustrated in Figure 2.2

2.3.2 Regular Grammar

A Regular Grammar (RG) (Chomsky, 1956), also called a **Type 3 Grammar** is a set
of recursive rewriting rules (called productions) that can be used to generate patterns
of strings that belong to a Regular Language. An RG can be recognized by a non-
deterministic Finite State Automaton (FSA) and it can either be right-regular or left-
regular. Every left RG generates exactly the language that some non-deterministic FSA
generates. In addition, every right RG is the reverse of some left RG and also generates
exactly the language that some non-deterministic FSA generates (the reverse FSA of
the left RG one). Finally, every RG is also a Context Free Grammar (see Section 2.3.3).

A right RG contains only productions of the following forms:

1. \( A \rightarrow \alpha \), where \( A \) is a non-terminal and \( \alpha \in \Sigma \).

2. \( A \rightarrow \alpha \Gamma \), where \( A \) and \( \Gamma \) are non-terminals and \( \alpha \in \Sigma \).

3. \( A \rightarrow \varepsilon \), where \( A \in N \).

A left RG contains only productions of the following forms:

1. \( A \rightarrow \alpha \), where \( A \) is a non-terminal and \( \alpha \in \Sigma \).

2. \( A \rightarrow \Gamma \alpha \), where \( A \) and \( \Gamma \) are non-terminals and \( \alpha \in \Sigma \).

3. \( A \rightarrow \varepsilon \), where \( A \in N \).

2.3.3 Context Free Grammar

A Context Free Grammar (CFG) (Chomsky, 1956), also called a **Type 2 Grammar**
is a set of productions that can be used to generate patterns of strings that belong to
a Context Free Language. A CFG can be recognized by a non-deterministic push-
down automaton (an automaton with a stack) and they are the theoretical basis for the
phrase structure of most programming languages (although a Context Sensitive Gram-
mar might be need for name resolution when variables with the same name appear in
different scopes).
2.3. Grammars

A CFG production is a rule of the form \( A \rightarrow \gamma \), where \( \gamma \) is a sequence of terminal and non-terminal symbols. An empty sequence is also allowed and in this case the production is called an epsilon (\( \varepsilon \)) production or a deletion rule. A rule rewrites (or replaces) non-terminal symbols on the left side of the production that appear in a string with other non-terminal or terminal symbols on the right side of the production.

Given a CFG, a string of terminal symbols can be generated by applying the following steps:

1. Start with a string that consists of only the start symbol.
2. Apply one of the productions in which the start symbol appears on the left hand size and replace it with the right hand side of the production.
3. Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all non-terminal symbols have been replaced by terminal symbols.

2.3.4 BNF

A Backus-Naur Form (BNF) (Knuth, 1964) is a formal notation used to describe the syntax of a Chomsky CFG. In this work, a BNF with regular extensions (contains regular expression operations) will be used. A BNF description can be found in the specification of many programming languages. A BNF defines classes of symbols whose names are enclosed in angle brackets (e.g., \( \langle \text{FilterExpression} \rangle \)), this notation is used for every non-terminal symbol. Terminals are literals (e.g., “$” or “:FilterOp”).

Every BNF rule is of the form: \( \langle \text{FilterExpression} \rangle ::= \langle \text{Relop} \rangle \text{ “ : FilterOp”} \).

The ‘::=’ symbol means that the non-terminal symbol on the left side of the rule can be replaced with the symbols appearing in the right side. The right side of the rule is called an expansion. Some non-terminals are able to be expanded using different expansions, in that case the ‘|’ symbol can be used to indicate different expansions. Figure 2.3 illustrates an example of a simple BNF.

2.3.5 Regular Tree Grammar

A Regular Tree Grammar (RTG) is a grammar that describes a set of directed trees. It is the tree equivalent of an RG (and a generalization of it) but it is very similar in appearance to a CFG, with the added notion of states. In fact, a CFG is special case of an RTG (May and Knight, 2006). CFGs are thought to represent string languages.
However, they can also be thought as representing tree languages, if for every string in the language, the derivation tree used to form the string is maintained (May and Knight, 2008).

As mentioned before, an RTG looks a lot like a CFG. It has start symbols, which are called states and rules (productions). An RTG rule has a single state on the left, and a tree on the right (May and Knight, 2008). The process of performing a derivation starts with a start state. Then it repeatedly erases states and in their place writes down the right sides (trees) of the rules used, until there are no states left to replace (May and Knight, 2008).

The CFG grammar of Figure 2.2 is a CFG grammar. This grammar recognizes the sentences:

- the cat barks.
- the dog meows.

In fact, it is not possible to create a CFG that recognizes the other two sentences of the grammar of Figure 2.2 but not these two as well. However, there can be an RTG that recognizes only the other two sentences. To formalize that, every context free language is the yield language of some regular tree language and the derivation trees of a CFG from an RTL. However, an RTL might not be the set of derivation trees from any CFG (unless re-labaleing is done) (Knight and Graehl, 2005). Figure 2.4 illustrates such an RTG as well as the language represented by it.

### 2.3.6 Synchronous Grammars

#### 2.3.6.1 Synchronous Context Free Grammar

A Synchronous Context Free Grammar (SCFG) (Aho and Ullman, 1969) is a generalization of CFG that simultaneously generates strings in two languages (Chiang, 2006). Their original service was the compilation of programming languages (Aho and Ullman, 1969), but they have also been used extensively in syntax-based Machine Translation (MT) (Yamada and Knight, 2001; Chiang, 2007; Graehl et al., 2008).

The difference from a CFG is that its productions have two right sides, which are called the source and target size respectively. The two right sides must have a relation (e.g., the source could be a grammar for English and the target one for French). When a derivation is performed using an SCFG the left side is rewritten as a pair of sequences of terminal and non-terminal symbols instead of one sequence of such.
2.4. Tree Transducers

Figure 2.5 illustrates an SCFG for the English sentence “the blue cat” and its Welsh translation “y gath glas”, along with the derivation that translates the English sentence to Welsh.

2.3.6.2 Synchronous Tree Substitution Grammar

A synchronous tree substitution grammar (STSG) is a collection ordered pairs of aligned elementary trees, which may be combined into a derived pair of trees (Eisner, 2003). Eisner (2003) define an elementary tree as a tuple \( \langle V, V^i, E, l, q, s \rangle \), where \( V \) is a set of nodes, \( V^i \subseteq V \) is the set of internal nodes, \( E \subseteq V^i \times V \) is a set of directed edges, and \( V^f = V - V^i \) is the set of frontier nodes, which all of them are leaves of the elementary trees. \( l : (V^i \cup E) \rightarrow L \) is a function that labels each internal node or edge, \( Q \) is a set states, \( q \in Q \) is the root state, and \( s : V^f \rightarrow Q \) assigns a frontier state to each frontier node. The leaves of the elementary trees can be either non-terminal symbols or terminal symbols (Chiang, 2006).

An STSG derivation starts with a pair of elementary trees (one for the source and one for the target language), which are rooted in the corresponding start symbol. Each production is a pair of elementary trees, and the leaf non-terminals are linked just as in a synchronous CFG (Chiang, 2006). The derivation proceeds by repeatedly choosing a leaf non-terminal symbol \( X \) and attaching to it an elementary tree rooted in \( X \).

2.4 Tree Transducers

According to Jones et al. (2012) a common approach for semantic parsing (Section 2.1) is to assume a tree structure to the natural language, or the meaning representation or both (Ge and Mooney, 2005; Kate and Mooney, 2006; Wong and Mooney, 2006). A tree transducer (Rounds, 1970; Thatcher, 1970) is a generalization of a finite state machine that operates on trees. The difference is that instead of accepting or rejecting a string, a tree transducer also transforms the input and generates a new one. Tree transducers have been successfully used in syntax-based MT (Yamada and Knight, 2001; Chiang, 2007; Graehl et al., 2008) and are suited to formalize tree transformation based models. A tree transducer can be defined using the STSG formalization of Section 2.3.6.2, and thus it is later used in Chapter 3 to describe the tree transducer system used in this work.
\[ S \rightarrow \text{NP} \text{ VP} \]
\[ \text{NP} \rightarrow \text{Det} \text{ NN} \]
\[ \text{Det} \rightarrow \text{the} \]
\[ \text{NN} \rightarrow \text{cat} \]
\[ \text{NN} \rightarrow \text{dog} \]
\[ \text{VP} \rightarrow \text{V} \]
\[ \text{V} \rightarrow \text{meows} \]
\[ \text{V} \rightarrow \text{barks} \]

(a) Example of a grammar.

(b) Derivation 1

(c) Derivation 2

(d) Derivation 3

(e) Derivation 4

Figure 2.2: An example grammar and its possible derivations.

\[ \langle \text{ScalarConstant} \rangle := \langle \text{Currency} \rangle ? \langle \text{Digit} \rangle^+ ( . \langle \text{Digit} \rangle^+ )? \]

\[ \langle \text{Currency} \rangle := \$ | € | £ \]

\[ \langle \text{Digit} \rangle := [0-9] \]

Figure 2.3: A simple BNF grammar example.
2.4. Tree Transducers

\[ q \rightarrow S(np1 \ vp1) \]
\[ q \rightarrow S(np2 \ vp2) \]
\[ np1 \rightarrow NP(dt \ nn1) \]
\[ np2 \rightarrow NP(dt \ nn2) \]
\[ vp1 \rightarrow VP(v1) \]
\[ vp2 \rightarrow VP(v2) \]
\[ dt \rightarrow Det(dtword) \]
\[ dtword \rightarrow \text{the} \]
\[ nn1 \rightarrow NN(nnword1) \]
\[ nnword1 \rightarrow \text{cat} \]
\[ nn2 \rightarrow NN(nnword2) \]
\[ nnword2 \rightarrow \text{dog} \]
\[ v1 \rightarrow V(vword1) \]
\[ v2 \rightarrow V(vword2) \]
\[ vword1 \rightarrow \text{meows} \]
\[ vword2 \rightarrow \text{barks} \]

(a) An RTG grammar example.

(b) RTG derivation 1

(c) RTG derivation 2

Figure 2.4: Example of an RTG grammar and the language recognized by it.
Chapter 2. Background

(a) An SCFG grammar example.

S \rightarrow \langle NP, NP \rangle
NP \rightarrow \langle Det Adj NN, Det NN Adj \rangle
Det \rightarrow \langle the, y \rangle
Adj \rightarrow \langle blue, glas \rangle
NN \rightarrow \langle cat, gath \rangle

(b) SCFG derivation step 1
(c) SCFG derivation step 2
(d) SCFG derivation step 3
(e) SCFG derivation step 4

(f) SCFG derivation step 5
(g) SCFG derivation result

Figure 2.5: Example of an SCFG, its derivation steps, and the full derivation.
Chapter 3

Machine Translation as Tree Transduction

In this chapter we introduce our method and give a detailed explanation of the parts it consists of. We have chosen to base and build our method upon Tree Transducer Toolkit (T3) (Cohn and Lapata, 2009), which learns a synchronous tree substitution grammar (STSG). The STSG allows local distortion of the tree topology and can thus naturally capture structural mismatches. Additionally, it allows translation pairs of non-isomorphic trees. This model was originally applied to monolingual translation (i.e., sentence compression) and not to semantic parsing.

Our method is novel compared to previous semantic parsing methods that employ a tree transducer for three main reasons. The first is that it can do tree-to-tree translation (tree-to-string is also supported) and not only tree-to-string (Yamada and Knight, 2001; Eisner, 2003; Jones et al., 2012), thus ensuring proper syntax of its output. The second one, is that it is based on a discriminative model instead of a generative model that has been used for the same or similar tasks in relevant work (Jones et al., 2012), which offers the ability to use many different types of features and on different levels of the derivation (see Section 3.4). For example, the generative model of (Jones et al., 2012) extracts a PCFG. The third and final one is that the methodology does not need to change in order to perform the task on the reverse direction (switching the source and target language), or to perform other tasks such monolingual translation or machine translation of between two natural languages. To perform the reverse task we could either just switch the source and target trees in the extracted grammar rules or switch the source and target languages and rerun the method. A third and interesting way would be to simultaneously extract rules for both directions and jointly train the
tree transducers. Lastly, the method assumes that both the source and target language are accompanied by a parse tree or an AST.

The major parts of the method are:

1. Extraction of word alignments. This step aims to identify word pairs in the yield of the source and target language respectively, which have some correspondence in meaning (usually one translates to the other). These pairs are used in the next part of the system in order to extract translation rules.

2. Grammar rules extraction. Using the word alignments of the previous step, a grammar of translation rules is extracted.

3. Language model extraction. A language model for the target side is extracted and is later used as a feature in order to guide the decoder into choosing proper structured, grammatical, and coherent translations.

4. Feature extraction. Several features for each grammar rule are extracted that are used in order to score derivations.

5. Model training. This step utilizes the training data to learn weights for the features in order to score correct derivations better than incorrect ones.

6. Decoding. The final step uses a chart-based dynamic programming algorithm, coupled with beam-search and a cube-pruning heuristic (Chiang, 2007) in order to approximate the best possible derivation for a given source tree.

### 3.1 Word Alignments Extraction

A very crucial element, and the very basis of the method, are word alignments. These are used to extract the translation rules that constitute the grammar. Word alignments with low alignment error rate (AER) are very important for this task because the quality of the alignments greatly influences that of the extracted grammar rules. We cannot measure the quality of the alignments we have extracted for our data because we have no annotated data and on some cases it might be difficult even for humans to agree on a gold alignment.

To perform word alignment extraction we have utilized the unsupervised version of the Berkeley aligner (Liang et al., 2006) instead of GIZA++. Many MT systems use
GIZA++ as a black box and usually select IBM Model 4 because of its good compromise between alignment quality and efficiency. Liang et al. (2006) implemented the Berkeley aligner in order to address the three main issues that people face with complex models like IBM Model 3 (Brown et al., 1993) and above. Firstly, these models are very slow during training time for a large corpus, even when highly optimized implementations like the ones in GIZA++ are used. Secondly, new fertility-based models like Model 6 offer modest improvements (Och and Ney, 2003) and are too complex to reimplement. Finally, these models are asymmetric, and symmetrization is commonly employed to improve alignment quality by intersecting the alignments induced in each translation direction (Och and Ney, 2000; Koehn et al., 2007).

Liang et al. (2006), motivated by this, observed that intersecting the predictions of two directional models outperforms each model alone (issue 3). These means that the predictions of the models for the two directions agree at test time. Their model extends this idea by also encouraging agreement of two HMM models during training time. However, experimental results show that their improvement in AER results in modest improvements in BLEU score (Papineni et al., 2002).

In a follow-up work, Denero (2007) created a similar but unsupervised word alignment HMM model that incorporates a syntax-aware distortion component, conditioned on the target language parse trees. Using these trees does not rule out any alignments, but rather influences the probability of transitioning between alignment positions. Each HMM is initialized by first jointly training IBM Model 1 on both directions. This model generates alignments that respect more the parse tree upon which they are conditioned and without making a sacrifice in alignment quality. Their method outperforms the GIZA++ implementation of IBM Model 4 and also reduces the number of aligned interior nodes by 56%, which measures the agreement between the alignments and the parses. These result in alignments that yield more rules for a tree transducer. For this reason, and also because no aligned training data are necessary, we have chosen to use this method as our word alignment extraction algorithm. We have used 10 iterations for both the Model 1 initialization and the training of the syntax model. An example of two aligned sentences is illustrated in Figure 3.1.

### 3.2 Grammar Extraction

After word alignments have been extracted, we can feed them, combined with our parallel tree training data to T3 in order to extract an STSG of translation rules.
Chapter 3. Machine Translation as Tree Transduction

Figure 3.1: Example of a predicted many-to-many word alignment for an instance of the Geoquery dataset.

Source sentence: How many people live in StateName_new_mexico?
Target sentence: answer population_1 stateid new_mexico

3.2.1 Translation Rules

This section describes the automatic method that harvests a grammar of translation rules from a parsed and word-aligned parallel training corpus. The word alignment is used to create one on the constituent level between nodes in the source and target trees. These pairs of aligned subtrees are next generalized to create tree fragments (also called elementary trees) from which the grammar rules are formed. Each rule uses the following template: \( \langle X, Y \rangle \rightarrow \langle \alpha, \gamma, \sim \rangle \), where \( X \) is the non-terminal root of the source subtree, \( Y \) is the non-terminal root of the target subtree, \( \alpha \) and \( \gamma \) are elementary trees rooted on \( X \) and \( Y \) respectively, and \( \sim \) is the one-to-one alignment between the frontier nodes in \( \alpha \) and \( \gamma \). Because some of the rules referenced later are quite big, the alignment will be skipped, when the rule is presented.

The translation rules are harvested using the algorithm of (Cohn and Lapata, 2009) that creates the constituent alignment, which is defined as the set of source and target constituent pairs whose yields are aligned to one another under the word alignment. The algorithm is based on the alignment template method of (Och and Ney, 2004), which creates a phrase-level alignment from a word level one. The aligned phrase pairs extracted for this method must satisfy three constraints. Firstly, at least one word
3.2. Grammar Extraction

Figure 3.2: Constituent alignment example from Cohn and Lapata (2009).

in one of the phrases must be aligned to a word in the other. Secondly, no word in either phrase must be aligned to a word outside the other phrase. Lastly, these phrases must be syntactic constituents. Equation 3.1 shows the definition of a constituent alignment. \( \nu_S \) and \( \nu_T \) are the corresponding source and target subtrees, \( A = (s, t) \) is the set of word-indices pairs (word alignments), \( Y(\cdot) \) produces the minimum and maximum word index for a subtree and \( \forall \) is the exclusive-or operator. Figure 3.2 illustrates an example from (Cohn and Lapata, 2009) of the word alignment and the corresponding constituent alignment for the sentence pair:

**Source sentence:** exactly what records made it and which ones are involved.

**Target sentence:** what records are involved.

\[
C = \{(\nu_S, \nu_T), (\exists (s, t) \in A \land s \in Y(\nu_S) \land t \in Y(\nu_T)) \land \\
(\exists (s, t) \in A \land s \in Y(\nu_S) \land \forall t \in Y(\nu_T))\} \tag{3.1}
\]

The algorithm continues by performing the aligned subtrees pairs generalization. This is achieved by replacing aligned child subtrees with variable nodes. Before that, we give an example of why this step is essential. We again use the example from (Cohn and Lapata, 2009). Consider the pair of aligned subtrees \([\hat{S} \text{ which ones are involved }]\) and \([\text{VP are involved}]\) from Figure 3.2. Using these we could extract the rule illustrated
in Equation 3.2. However, such a rule is not very useful since it only matches a very specific source tree and will not generalize well across unseen ones. This problem of the original model is also present in our case. In order to solve this problem, the algorithm extracts the most general (minimal) rules. This is achieved by replacing a subtree with a variable while still honouring the word-alignment. The most general rule for the one in Equation 3.2 is shown in Equation 3.3, where indices indicate the alignment. Additionally, the algorithm also allows a limited number of generalizations to be skipped during the extraction process by setting the recursion depth parameter. For example, if we increase the recursion depth by one we will also extract the rules shown in Equation 3.4.

\[
\langle \bar{S}, VP \rangle \rightarrow \langle [\bar{S} [WHNP [WP which]] [S [NP [NNS ones]][VP [VBP are] [VP [VBN involved]]]],
\]
\[
[VP [VBP are] [VP [VBP involved]]] \rangle
\] (3.2)

\[
\langle \bar{S}, VP \rangle \rightarrow \langle [\bar{S} WHNP_{\epsilon} S_1], VP_1 \rangle
\] (3.3)

\[
\langle \bar{S}, VP \rangle \rightarrow \langle [\bar{S} WHNP_{\epsilon} W P_{\epsilon} S_1], VP_1 \rangle
\]

\[
\langle \bar{S}, VP \rangle \rightarrow \langle [\bar{S} WHNP_{\epsilon} [S NP_{\epsilon} VP_1]], VP_1 \rangle
\] (3.4)

### 3.2.2 Copy Rules

Although the algorithm presented in Section 3.2.1 can extract rules with good generalization ability from the training data, Cohn and Lapata (2009) state that there is no guarantee that the extracted rules will have good coverage on unseen trees. Example instances of this problem are tree fragments that contain previously unseen terminals or non-terminals, or an unseen sequence of children for a parent non-terminal. For all the above cases, the transduction algorithm will not succeed because it will be unable to cover the source tree. To address this problem Cohn and Lapata (2009) added copy rules to the grammar which copy parts of the source tree to the target tree. However, in our case this type of rules is not useful in our task. The only cases that this type of rules would be useful in our case are terminals that happen to be the same in the source and target language since most non-terminals are different in the two languages. For this reason we have chosen to disable this set of rules in the system.
3.2.3 Epsilon Rules

Another set of rules used in the system are deletion rules. These rules delete parts of the source tree. A set of rules is created for each source CFG production, this allows the deletion of subsets of the child nodes. The aim of the deletion rules is to attempt to preserve the most important child nodes. The importance is measured using the head-finding heuristic of the Collins’ parser (Collins, 2003). For more details about this process refer to (Cohn and Lapata, 2009).

3.3 Language Model

For the purpose of guiding the decoder we build a bigram language model for the yield (terminals) of the target language using the exact same process and parameters as in Section 5.2. The conditional log-probability of the bigram model is used as a feature in the decoder instead of a trigram, which was used in the original work. We chose a bigram instead of a trigram because we do not have enough training data to be able to build a decent trigram model. For more information on the decoder see Section 3.5.

3.4 Feature Extraction

In addition to an LM, the model of (Cohn and Lapata, 2009) define a feature space over source trees, \( x \) and target derivations \( d \). A derivation consists of a series of grammar rules that produce the target tree from the source tree, and it contains both the source and target trees. However, the extracted features are defined over grammar rules because weighted synchronous grammars allow only features that decompose with the derivation. The derivation features are defined as the summation of the features of all the rules used in the derivation. For more information on the scoring function for a derivation see Section 3.5.

3.4.1 T3 Features

All the features described in this section are boolean (i.e., their value can be either 0 or 1) and were used in (Cohn and Lapata, 2009).

Type: A total of three features which model the source of the rule. Whether it was harvested from the training set or it was created as a copy or deletion rule.
**Root:** Three features. The root category of the source subtree, \( X \), the root category of the target subtree, \( Y \), and the root categories of their conjunction, \( X \land Y \).

**Identity:** Four features are extracted that match the source and target subtrees. The existence of the source side \( \alpha \) in the rule, the existence of target side \( \gamma \), the existence of both, and the equality of \( \alpha \) and \( \gamma \).

**Unlexicalised Identity:** A replication of the above features but with any terminals that appear in both trees removed from their frontiers.

**Rule Count:** This feature is always one (and in contrast with other features, it is the same for all the rules). It is used to count the number of rules that are used in a derivation.

**Word Count:** Two features that count the number of terminals in \( \alpha \) and \( \gamma \) respectively. These are the same features for each rule (but with different values).

**Yield:** These features can be further split into three categories. The first one contains only one feature that checks if \( \alpha \) and \( \gamma \) contain the same terminals (the order of the terminals is not important though) and is the same feature for all rules. The second category uses identity for each terminal that appears in both \( \alpha \) and \( \gamma \) and the third for those that only appear in \( \alpha \).

**Length:** Two features are used. The first one records the difference in the number of nodes of \( \alpha \) and \( \gamma \), whilst the second one checks if \( \gamma \) has less nodes than \( \alpha \).

**Number of Variables:** Two features that count the number of variables in \( \alpha \) and \( \gamma \) respectively. These are the same features for each rule.

### 3.4.2 New Features

The features of Cohn and Lapata (2009) were built for the task of sentence compression. For this reason, some of them do not fit well in an MT task. For example, the features that belong in the second category of yield features will always be 0 for all source words, and for all the rules (except of the very rare case that a source word translates to the same one in the target language). This results in having a feature of the third category for every source terminal in the source side. This is not reasonable as the second category was created to model words that remained in the sentence during the compression task and the third to model words that were erased. In our case all...
source side terminals except those that translate to the same word will be considered as deleted words, which obviously is not true.

Another problem of the original model is that these features do not take into account the structure of the tree, or with which other rules they were used. The source of this problem is that these features are not defined over derivations but over rules and they are summed up in order to create features for the derivation. Moreover, there are no task specific features in this model. For instance, in the Excel commands translation task the decoder could make use of features that record the type of a column. One of the common errors is that the decoder uses the wrong type for a column (e.g. a column of numbers is considered a column of currencies). However, unless features on the derivation error are defined, it is not possible to create a feature for this because the same rule that creates a column for the target size could be used for different types of columns. Lastly, this modification would also allow the use of traditional MT features that are present in an MT decoder.

3.5 Training the Model

In this section, the model that is learned during training time to score the derivations that can be created from given source trees will be explained. The grammar extracted in Section 3.2 has the capability of creating multiple possible target trees from a source tree. There is also the case that the same target tree can be generated using different derivations, this phenomenon is called spurious ambiguity (Cohn and Lapata, 2009). Fully addressing this ambiguity requires to aggregate all the derivations that produce an identical target tree and would additionally make the problem NP-complete (Knight, 1999). Instead, the derivations are scored in order to learn weights $w$ for the features employed to rank derivations that produce the correct target tree (reference target tree) or the best possible target tree higher than others by assigning them higher scores. To accomplish this Cohn and Lapata (2009) define a scoring function over derivations $d$ and feature weights $w$ (see Equation 3.5).

$$
score(d;w) = \langle \Psi(d),w \rangle
$$

(3.5)

$\Psi$ is a function that returns the features for a derivation and $\langle \cdot, \cdot \rangle$ returns the inner product of two vectors. The definition of $\Psi$ is given in Equation 3.6.

$$
\Psi(d) = \sum_{r \in d} \phi(r, source(d)) + \sum_{m \in ngrams(d)} \psi(m, source(d))
$$

(3.6)
Chapter 3. Machine Translation as Tree Transduction

The derivation rules are represented by $r$, $ngrams(d)$ represents the set of ngrams in the yield of the target tree, $m$ is used to sum over these ngrams and $\psi$ is a feature function over them. The second summand in (Cohn and Lapata, 2009) is the language model. However, the model also allows the definition of a different function $\psi$ over these ngrams, as $\psi$ is conditioned on the source tree of $d$ (source($d$)). $\phi$ is a function that is used to return the feature vector of each rule. This function is also conditioned on the source tree. As a result, the feature function $\Psi$ is also conditioned over the source tree. As mentioned before, traditional STSGs only allow features that decompose with the derivation. These features are represented by the first summand and are applied to rules and not terminals or ngrams of terminals. On the other hand, the language model cannot be decomposed, it is applied on ngrams of terminals instead of rules and it is represented with the second summand. Any new features added to $\Psi$ that cannot be decomposed (e.g., features for the final target tree) should be added as another summand in Equation 3.6. Cohn and Lapata (2009) avoided the definition of such features since the increase the complexity of inference for training and decoding. For example, the use of the language model (or any function over ngrams) results in a grammar that is the intersection between the ngram language and the original grammar (Chiang, 2007).

The objective of the training procedure of (Cohn and Lapata, 2009) is to find the parameters $w$ that for all training instances the reference derivation gets at least as high score as any other derivation. It is also important to know how much a predicted target tree differs from the reference one. This difference can be measured in different ways. This can be expressed by using different loss functions $\Delta(y_i, y)$, where $y_i$ is reference tree for the $i^{th}$ training instance and $y$ is the predicted one.

3.5.1 Loss Functions

Cohn and Lapata (2009) have built a total of 17 different loss functions in T3 that can be used to measure the accuracy of predictions. However, other loss function than these could also be used (e.g., a loss function that is the weighted sum of other loss functions). A loss function is defined over derivations, and thus can use any of the elements in the derivation. This includes tokens, ngrams, and CFG rules. A brief explanation of some of the loss functions follows. The full list of all 17 loss functions is given in Appendix A.

Unordered token errors with length penalty: This loss function counts the number
of tokens in the yield of the predicted derivation that do not appear in the yield of the gold one. It also employs a length penalty for short output (the prediction has less tokens than the derivation).

**Unordered token precision * brevity penalty (BP1):** The number of tokens of the reference that were predicted multiplied by a brevity penalty called BP1 (Cohn and Lapata, 2007).

\[ BP1 = \exp(1 - \max(1, \frac{\ell}{x})) \]

**Asymmetric token hamming distance:** This loss functions calculates the hamming distance between unordered bags of tokens. It measures the number of tokens that did not appear in the reference, along with a penalty for short output.

\[ \Delta_{\text{hamming}}(d^*, d) = FP + \max(l - (TP + FP), 0) \]

Where, \( l \) is the length of the reference, \( TP \) and \( FP \) are the number of true and false positives predicted tokens, when the predicted tree \( d \) is compared to the reference tree \( d^* \).

### 3.5.2 Linear Model Training Framework

To train the linear model Cohn and Lapata (2009) use the framework of Tsochantaridis et al. (2005) for learning Support Vector Machines (SVMs) over structured output spaces. The implementation used is SVM\(_{\text{struct}}\)\(^1\). This framework offers regularization that can prevent overfitting of a discriminative model when a large number of features is used. The details of the optimization problem solved and the algorithm solving it are skipped since they are described in details in (Cohn and Lapata, 2009).

### 3.6 Decoding

The last step in our methodology is decoding. The objective of the decoder is to find the best scoring derivation for a target tree that can be created by the grammar rules given a source tree. The maximization objective of the decoder is illustrated in Equation 3.7.

\[ d^* = \arg\max_{d: \text{source}(d) = x} score(d; w) \quad (3.7) \]

The given source tree is \( x \), \( d \) is the derivation, \( \text{source}(d) \) returns the source tree of the derivation and \( score \) is the scoring function defined in Equation 3.5. Cohn and

\(^1\)http://svmlight.joachims.org/svm_struct.html
Lapata (2009) approximate this maximization problem using a chart-based dynamic programming algorithm. This algorithm is an extension of the inference algorithm of Eisner (2003) for weighted STSGs.

The algorithm uses a chart $C$, the cells of which record the partial target tree with the best score for each source node and uses a post-order traversal to process the source tree. In order to record the maximizing rule back-pointers, $B$ are used to store the child chart cells that fill each variable in the rule. To allow the scoring of ngram features, the chart is also indexed by the n-1 terminals at the left and right edges of the target tree’s yield. Lastly, it must be noted that the decoding algorithm does not perform exact inference but uses a beam-search and a cube-pruning heuristic (Chiang, 2007) that approximates the conditional log-probability of the ngram language model used with that of a unigram model in order to make decoding feasible. A more detailed analysis of the algorithm and why exact decoding is infeasible can be found in (Cohn and Lapata, 2009).
Chapter 4

Datasets

We will now describe the datasets that were used in our experiments. We have experimented on two datasets. The first one, is an extended version of the NLyze dataset and the second one, is Geoquery.

4.1 Excel Commands

4.1.1 Description

The basis of this dataset is seven Excel spreadsheets provided by the Excel product team. These come from conceptually different areas such as employee payrolls, inventory management, country facts and sales invoices. This difference in the sheets results in a more variable vocabulary and relations between the sheet elements.

For six of these spreadsheets they constructed 10 tasks per spreadsheet. Each task is one that can be solved by a single Excel formula, e.g., =MAX(IF(“xerten”=Table1[item], Table1[amt_due], “’”)). The tasks involve lookup tasks, reduce/selection operations, arithmetic formulas, and combinations of these operations. For the seventh one they created only 4 tasks. This resulted in a total of 64 different tasks.

In order to create the natural language descriptions for each task, they took before/after images of the spreadsheet. An online crowd-sourcing platform was utilized to ask users to look at the before/after images and describe what they would tell a human to do in order to accomplish the illustrated task. An example of before/after images for one of the tasks is illustrated in Figure 4.1. Then, they cleaned the user provided data to ensure that each described task actually matched the target task. Additionally, they removed any typos, and ensured that the commands used column names
or Excel notation when referring to data in the sheet. This process resulted in a total of 4401 natural language descriptions each one mapped to a specific task and spreadsheet.

### 4.1.2 Domain Specific Language (DSL)

According to Mernik et al. (2005), Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. This is in opposition to a general-purpose language (GPL), which is broadly applicable across domains, and lacks specialized features for a particular domain. DSLs vary extensively and range from widely used languages for common domains, such as HTML for web pages, down to languages used by only a single piece of software. DSL development is hard, requiring both domain knowledge and language development expertise.

Gulwani and Marron (2014) designed a DSL that is expressive enough to express desired categories of tasks and restrictive enough to allow effective translation from varied descriptions in natural language, and to allow for a simple end-user friendly interaction model. Their DSL supports compositions of some basic forms of map, filter, and reduce operators. They focused on common spreadsheet tasks which were extracted by observing with which tasks end users struggle with in Excel help forums. These tasks include conditional filtering (e.g., selection of cells of column that are greater than a certain value), conditional arithmetic (e.g., addition of two columns), and lookup tasks (e.g., find maximum value of a column). In addition to these it supports the selection or formatting of spreadsheet cells.
4.1. Excel Commands

4.1.2.1 BNF Definition

A small description of the DSL is described in the paper but an implementation is not available. For this reason we have implemented its BNF using SableCC (Gagnon and Hendren, 1998). We have excluded parentheses in the shown definition of the grammar and we assume that the reader is familiar with the form of a BNF. This information is captured by the abstract syntax tree (AST) because each function always gets the same number of arguments. For instance And always gets two arguments (two FilterExpressions).

\[
\text{⟨Program⟩ } := \text{ ⟨VectorExpression⟩} \\
\text{ | ⟨ScalarExpression⟩} \\
\text{ | ⟨Format⟩}
\]

\[
\text{⟨QueryExpression⟩ } := \text{ ⟨SelectRows⟩ ⟨RowSourceExpression⟩ ⟨FilterExpression⟩ :null} \\
\text{ | ⟨SelectCells⟩ ⟨ColumnName⟩ ⟨RowSourceExpression⟩ ⟨FilterExpression⟩ :null} \\
\text{ | ⟨SelectRows⟩ ⟨FilterExpression⟩ :null} \\
\text{ | ⟨SelectCells⟩ ⟨ColumnName⟩ ⟨FilterExpression⟩ :null}
\]

\[
\text{⟨RowSourceExpression⟩ } := \text{ GetTable Tbl} \\
\text{ | MTable} \\
\text{ | GetActive} \ | \text{ GetFormat Tbl ⟨FilterExpression⟩}
\]

\[
\text{⟨FormatExpression⟩ } := \text{ ⟨FormatFunc⟩+}
\]

\[
\text{⟨FormatFunc⟩ } := \text{ Color ⟨ScalarConstant⟩} \\
\text{ | FontSize ⟨ScalarConstant⟩} \\
\text{ | Bold ⟨BooleanConstant⟩} \\
\text{ | Italics ⟨BooleanConstant⟩} \\
\text{ | Underline ⟨BooleanConstant⟩}
\]

\[
\text{⟨FilterExpression⟩ } := \text{ ⟨Relop⟩ :FilterOp} \\
\text{ | And ⟨FilterExpression⟩ ⟨FilterExpression⟩ :FilterOp} \\
\text{ | Or ⟨FilterExpression⟩ ⟨FilterExpression⟩ :FilterOp}
\]
Chapter 4. Datasets

| Not \( \langle FilterExpression \rangle :FilterOp \)
| True: FilterOp

\( \langle ScalarExpression \rangle := \langle Rop \rangle \)
| Count \( \langle RowSourceExpression \rangle \langle FilterExpression \rangle :FilterOp \)
| Count \( \langle FilterExpression \rangle :FilterOp \)
| \( \langle BopScalar \rangle \)
| Lookup \( \langle ScalarExpression \rangle \langle RowSourceExpression \rangle \langle ColumnName \rangle \langle ColumnName \rangle \)
| \( \langle ScalarConstant \rangle \)
| \( \langle StringConstant \rangle + \)

\( \langle VectorExpression \rangle := \langle BopVector \rangle \)
| \( \langle ColumnName \rangle \)
| Lookup \( \langle ColumnName \rangle \langle RowSourceExpression \rangle \langle ColumnName \rangle \langle ColumnName \rangle \)

\( \langle BopScalar \rangle := \text{Add} \langle ScalarExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Sub} \langle ScalarExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Mult} \langle ScalarExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Div} \langle ScalarExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)

\( \langle BopVector \rangle := \text{Add} \langle VectorExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Sub} \langle VectorExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Mult} \langle VectorExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Div} \langle VectorExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Add} \langle VectorExpression \rangle \langle ScalarExpression \rangle \langle ResultType \rangle \)
| \text{Sub} \langle ScalarExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Mult} \langle ScalarExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Div} \langle ScalarExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Add} \langle VectorExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Sub} \langle VectorExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Mult} \langle VectorExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
| \text{Div} \langle VectorExpression \rangle \langle VectorExpression \rangle \langle ResultType \rangle \)
\[ \langle \text{Rop} \rangle \quad ::= \quad \text{RSum} \langle \text{ColumnName} \rangle \langle \text{RowSourceExpression} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RAverage} \langle \text{ColumnName} \rangle \langle \text{RowSourceExpression} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RMin} \langle \text{ColumnName} \rangle \langle \text{RowSourceExpression} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RMax} \langle \text{ColumnName} \rangle \langle \text{RowSourceExpression} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RSum} \langle \text{ColumnName} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RAverage} \langle \text{ColumnName} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RMin} \langle \text{ColumnName} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
| \quad \text{RMax} \langle \text{ColumnName} \rangle \langle \text{FilterExpression} \rangle \langle \text{ResultType} \rangle \\
\]

\[ \langle \text{Relop} \rangle \quad ::= \quad \text{LT} \langle \text{ColumnName} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{GT} \langle \text{ColumnName} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{EQ} \langle \text{ColumnName} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{LTEQ} \langle \text{ColumnName} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{GTEQ} \langle \text{ColumnName} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{LT} \langle \text{ColumnName} \rangle \langle \text{ScalarExpression} \rangle \\
| \quad \text{GT} \langle \text{ColumnName} \rangle \langle \text{ScalarExpression} \rangle \\
| \quad \text{EQ} \langle \text{ColumnName} \rangle \langle \text{ScalarExpression} \rangle \\
| \quad \text{LTEQ} \langle \text{ColumnName} \rangle \langle \text{ScalarExpression} \rangle \\
| \quad \text{GTEQ} \langle \text{ColumnName} \rangle \langle \text{ScalarExpression} \rangle \\
| \quad \text{LT} \langle \text{ScalarExpression} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{GT} \langle \text{ScalarExpression} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{EQ} \langle \text{ScalarExpression} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{LTEQ} \langle \text{ScalarExpression} \rangle \langle \text{ColumnName} \rangle \\
| \quad \text{GTEQ} \langle \text{ScalarExpression} \rangle \langle \text{ColumnName} \rangle \\
\]

\[ \langle \text{ScalarConstant} \rangle \quad ::= \quad \langle \text{Currency} \rangle ? \langle \text{Digit} \rangle + ( . ? \langle \text{Digit} \rangle + )? \]

\[ \langle \text{Currency} \rangle \quad ::= \quad \$ | \€ | \£ \]

\[ \langle \text{Digit} \rangle \quad ::= \quad [0-9] \]
\[
\langle \text{StringConstant} \rangle := \langle \text{EnglishLetter} \rangle + \\
\langle \text{BooleanConstant} \rangle := \text{True} \mid \text{False} \\
\langle \text{ColumnName} \rangle := \langle \text{ColumnIndex} \rangle \langle \text{ResultType} \rangle \\
\mid \langle \text{ColumnRange} \rangle \langle \text{ResultType} \rangle \\
\langle \text{ColumnIndex} \rangle := \langle \text{EnglishLetter} \rangle + \langle \text{Digit} \rangle + \\
\langle \text{EnglishLetter} \rangle := [A-Za-z]
\]

### 4.1.3 Preprocessing

In this section we will describe the preprocessing steps that we have applied to the data.

1. We parse the English sentence using the Stanford parser (Klein and Manning, 2003; Socher et al., 2013). It must be noted that these parses contain errors. One reason for this is that the parser is not aware of column references (e.g., e1:e13, or column e, or totalpay) and sentences might be ungrammatical for the parser which is trained on news corpora.

2. We use our implementation of the DSL to create an AST for the corresponding Excel command. We have used the DSL version of the commands for two reasons. Firstly, there is no standard parser for Excel commands in order to retrieve an AST, while for the DSL this is possible. Secondly, Gulwani and Marron (2014) used the DSL version of the commands in their experiments. However, it would be interesting to create an Excel command parser and make a comparison of the results achieved when using each representation.

3. For each instance in the data we find the spreadsheet that it came from and load its contents in order to store the column names and the data contained in it.

4. We normalize both the English sentence and the Excel command using the following steps.

   (a) We find all currency tokens and replace them with the special token `cur`.
   (b) We find all numeric tokens and we convert them into the special token `numb`. 
(c) We find all column references and we convert them into the special token `col`.

(d) We find all string literal tokens that refer to a column's cell and we convert them into the special token `str`.

(e) For each type of token that we normalized, we append `'_1'` to the first one in the instance, `'_2'` to the second one, and so on. We also store the original value so that we can retrieve it and replace it back.

The normalized tokens are also replaced in the corresponding parse trees. An example of the normalization algorithm is visualized in Figures 4.2.

5. For each instance in the data we create an index that contains information about it. This includes the spreadsheet that it came from, the mapping used (which values were normalized into what), and each index in the dataset.

sum totalpay where over $4300

(a) Original English sentence

sum col_1 where over cur_1

(b) Normalized English sentence

Figure 4.2: Application example of the normalization algorithm on one of the Excel dataset instances.

RSum g2:g11 LT $4300 g2:g11 :Range<currency>

(a) Original Excel command

RSum col_1 LT cur_1 col_1 :Range<currency>

(b) Normalized Excel command

Figure 4.3: Application example of the normalization algorithm on one of the Excel dataset instances.

4.2 Geoquery

4.2.1 Description

Geoquery (Zelle, 1995) consists of a database written in Prolog that contains information about US geography and a set of 880 sentences in English. 250 of have also been translated into Spanish, Japanese and Turkish. The meaning of each sentence is
represented operationally as a query in a suitable database query language by an annotated Prolog logical form. This database is part of the Turbo Prolog 2.0 distribution (Borland International, 1988). Zelle (1995) describes the query language as a first-order logical form augmented with some higher-order predicates or meta-predicates, for handling issues such as quantification over implicit sets. It was not designed with any notion of appropriateness for representation of natural language in general, but rather as a direct method of compositionally translating sentences into unambiguous, logic-oriented database queries. Kate et al. (2005) developed a variable-free, functional query language (FUNQL) for Geoquery to allow for direct translation of the functional forms into first-order logical forms, which can be used to query the target database.¹ This variable-free language allowed semantic parsers that cannot deal with logical variables to be applied on the task (Ge and Mooney, 2005; Kate and Mooney, 2006; Ge and Mooney, 2009).

An example of a query in English and its equivalent Prolog query is demonstrated in Table 4.1.

<table>
<thead>
<tr>
<th>English sentence</th>
<th>What states does the Ohio river run through?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prolog query</td>
<td>$answer(x_1, (state(x_1), traverse(x_2, x_1), equal(x_2, riverid(ohio))))$</td>
</tr>
<tr>
<td>Functional query</td>
<td>$answer(state(traverse_1(river(riverid(ohio))))))$</td>
</tr>
</tbody>
</table>

**Table 4.1:** Geoquery instance example: Natural language question in English and its equivalent Prolog and functional queries.

The data can be downloaded as an XML file containing the natural language sentences, their syntactic parses in English, their representations in the two query languages and an annotation of the named entities that appear in each sentence. For instance, the sentence “What is the population of Tempe Arizona?” contains the following named entities:

**Tempe:** CityName

**Arizona:** StateAbbrev

In our experiments we will be using the functional version of the query language and the English natural language sentences because their syntactic parses which are crucial for our method are also available.

¹For a description of both the original and the functional query language for geoquery see [http://www.cs.utexas.edu/~ml/wasp/geo-funql.html](http://www.cs.utexas.edu/~ml/wasp/geo-funql.html)
4.2. Geoquery

Before we use the data in our experiments we have first applied several preprocessing steps.

1. We remove the ‘?’ or the ‘.’ at the end of the sentence because it does not exist in the syntactic parse and it creates noise during word alignment extraction.

2. Tokens in the natural language sentence that belong to the same named entity are first lower-cased and then merged into one using the ‘_’ character. For example ‘New York’ is transformed into ‘new_york’ and ‘Guadalupe Peak’ is transformed into ‘guadalupe_peak’. This allows us to get better word alignments, which are used to extract rules for our method (see Section 3.1 for more details on this). We also add the type of name entity to the start of the new token. Namely, ‘New York’ is transformed into ‘CityName_new_york’.

3. In a similar way to the previous step we also modify the syntactic parse of the sentence in order to match the new named entities. All the NNP and NNPS nodes that refer to the same entity are merged into one. As an example, the AST of the sentence ‘How high is Guadalupe Peak?’ is shown in Figure 4.4.

4. We create an AST for the code side. For each token we first assign a preterminal tag. This is done for two reasons. Firstly, this creates a structural similarity between the code and the natural language trees, and secondly, many of the libraries used in our method require the trees to be in Penn Treebank format (Marcus et al., 1994). All named entities are assigned the tag NNP. For each predicate we assign itself starting with a capital letter and with a ‘P’ appended at the end as its POS tag. However, according to Zelle (1995) some predicates...
(e.g., highest and lowest) are analogous. To model this, we have assigned them
to the same POS tag, the mapping we utilized is shown in Table 4.2. During
preliminary experiments, we noticed that the modelling of analogous predicates
resulted in a significant increase of system performance. Then, for each POS
tag we add a node as its predecessor (name of the predicate and an appended
T at the end). Finally, we add the tree that corresponds to the predicate of the
named entity or entities that it gets as argument as its rightmost child. The result
of this process for a sentence with code tokens ‘answer size largest city loc_2
stateid alaska’ is illustrated in Figure 4.5. Alternatively, instead of this process,
the meaning representation could have been used as a tree (Jones et al., 2012) as
shown on Figure 4.6. However, this results in only retrieving word alignments
(see Section 3.1) for the entities in the natural language sentence since these
are the only terminals in the code side, or even worse, to retrieving erroneous
word alignments that align entities in the code side to words that they often co-
occur with in the natural language. We noticed during preliminary experiments
that when these word alignments were used in our method (see Chapter 3), it
resulted in poor rule extraction (see Section 3.2) and in many cases the rule
extraction algorithm was even unable to extract any rules at all, thus resulting in
a system with 0% accuracy (see Section 6.2).

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Analogous Predicate</th>
<th>POS Tag</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>lowest</td>
<td>EstP</td>
<td>opposites</td>
</tr>
<tr>
<td>largest</td>
<td>smallest</td>
<td>EstP</td>
<td>opposites</td>
</tr>
<tr>
<td>longest</td>
<td>shortest</td>
<td>EstP</td>
<td>opposites</td>
</tr>
<tr>
<td>highest</td>
<td>largest</td>
<td>EstP</td>
<td>elevation</td>
</tr>
<tr>
<td>longest</td>
<td>largest</td>
<td>EstP</td>
<td>length</td>
</tr>
<tr>
<td>most</td>
<td>fewest</td>
<td>MaxminP</td>
<td>opposites</td>
</tr>
<tr>
<td>largest_one</td>
<td>smallest_one</td>
<td>Est_oneP</td>
<td>opposites</td>
</tr>
<tr>
<td>higher_2</td>
<td>lower_2</td>
<td>ErP</td>
<td>opposites</td>
</tr>
<tr>
<td>high_point_1</td>
<td>low_point_1</td>
<td>PointP</td>
<td>opposites</td>
</tr>
<tr>
<td>high_point_2</td>
<td>low_point_2</td>
<td>PointP</td>
<td>opposites</td>
</tr>
<tr>
<td>high_point_1</td>
<td>high_point_2</td>
<td>PointP</td>
<td>elevation</td>
</tr>
</tbody>
</table>

Table 4.2: Analogous predicates and their POS tag.
Figure 4.5: Creation of an AST from code tokens. Code tokens are shown in blue color.

Figure 4.6: Creation of an AST from code tokens. Code tokens are shown in blue color.
Chapter 5

Baselines

5.1 Retrieval Baseline

The first baseline that we implemented is the information retrieval baseline described in (Quirk et al., 2015). It is based on the fact that multiple users of a system might want it to perform the same operation and they might describe it in a similar way. It first stores the natural language descriptions of the programs that appear in the training set. To predict the program that matches a description in natural language for a test instance, it searches for the closest description in the training set and finally returns the associated program. This baseline can be built upon several different similarity metrics such as character or string edit distance, cosine similarity, hamming distance and Jaccard similarity. Quirk et al. (2015) report that string edit distance achieved the best results on a development set (they use a different dataset than this work). However, it is not clear if this was implemented on the token or character level. For this reason we implemented and measured the performance of both versions on our datasets. Lastly, Quirk et al. (2015) report that as the training data become more this baseline should increase in both quality and coverage.

5.2 Phrased-Based Machine Translation

The second baseline that we implemented was also used in (Quirk et al., 2015) but it was first introduced in Andreas et al. (2013). This baseline casts semantic parsing as phrase-based statistical machine translation (SMT). According to Quirk et al. (2015) a phrase-based SMT system (Och et al., 1999; Koehn et al., 2003) can be seen as incremental step beyond retrieval. This is because such a system segments the training
data and attempts to match and assemble those segments at runtime. If the phrase length is unbounded then retrieval is the special case in which for every instance its length is used as the phrase length.

The method starts by converting the parse trees or ASTs of the two languages (in our case natural language and its semantic parse) into flat sequences of tokens using a pre-order left-to-right traversal. This process is called linearisation and each token is also annotated with its arity (the number child nodes that it had in the parse tree or the AST) which allows us to reconstruct the tree, and also to check if predicted programs are well formed. An example of this process is illustrated in Figure 5.1.

![Figure 5.1: Conversion of a tree into a flat sequence of tokens using a pre-order left-to-right traversal (tree linearisation).](image)

The next step of the method is to feed the parallel corpus of linearised trees into a statistical machine translation system. We have chosen to use Moses (Koehn et al., 2007) as in (Andreas et al., 2013), but any similar system could be used instead. Additionally, we have allowed the phrase length limit to be greater than that of the phrases in the training set.

The method continues by extracting word alignments between the linearised representations of the source and target language for each instance. To perform word alignment extraction the implementation of IBM Model 4 (Brown et al., 1993) in GIZA++ (Och and Ney, 2003) is used. Alternatively, the Berkeley aligner (Liang et al., 2006) could be used instead. The alignment algorithm is run in both directions (natural language to code and code to natural language). Then, the resulting alignments are symmetrized using Moses implementation (Koehn et al., 2007) of the symmetrization algorithm described in (Och and Ney, 2000) to obtain many-to-many alignments.

The extracted alignments from the previous step are used to extract a phrase translation table. This table consists of phrases in natural language and their corresponding
phrase in code together with the translation probability. Each of these pairs of phrases are aligned source and target phrases.

Next step is the estimation of an ngram language model (LM) for the target language (code). A language model is very useful for the decoder as it guides the decoder into choosing translations with proper structure (for example it encodes information about which predicates should fit into another predicate) and it allows the modelling of local coherence and grammaticality. Language models are better when they are estimated on a large corpus. However, such a corpus is not available for the datasets we used. Following Andreas et al. (2013) we have used the training data to learn one. We have built a bigram LM using the SRILM (Stolcke, 2002) implementation of interpolated Witten-Bell discount (Bell et al., 1990).

The final step is to tune and use a decoder that maximizes the weighted sum of the probabilities of the translations (according to the phrase translation table) and the language model score. The decoder weights are tuned on a held out tuning set which makes up 5% of the data (see Chapter 6 for more details about the tuning phase and the structure of our experiments). Using the selected weights it produces a list of k-best translations. In our experiments we have used k=100 and we retain as output the first prediction in the list, that can be successfully converted into a well-formed program by using the arity of the tokens. It should be noted that this method was originally (Andreas et al., 2013) applied on the Geoquery dataset on pairs that consisted of a series of tokens in natural language and the meaning representations used in (Jones et al., 2012), that are trees the nodes of which have a maximum of two child nodes. Finally, the performance of this baseline and the quality of the produced translations could be improved by integrating this constraint into the decoder as proposed by (Quirk et al., 2015).
Chapter 6

Experiments

In this chapter the experiments performed alongside their results will be discussed.

6.1 Cross Validation and Tuning

Both of the datasets used for experiments are quite small. For this reason a 10-fold cross validation was used during the experiments. 5% of the data is held out for tuning the parameters of each model. The remaining 95% of the data is split into 10 parts. On each of the 10 folds, a different tenth of the data is used as the test set and the nine remaining are used as the training set and the model is tuned on the tuning set. We chose this architecture because we noticed a big variance on the results of our folds, during preliminary experiments that we performed.

The retrieval baseline does not have any parameters to tune and therefore does not make any use of the tuning set. For the phrased-based MT baseline, for each fold, the decoder weights were tuned using Moses’ (Koehn et al., 2007) implementation of minimum error training (MERT) (Och, 2003) on the tuning set. The NP lists used by both Jones et al. (2012) and Andreas et al. (2013) are added as instances in the phrased-based MT system and as rules in the tree transducer. Lastly, for the tree transducer system the following parameters are tuned by evaluating the system’s performance on the tuning set:

1. The loss function (1-17, see Section 3.5.1).

2. The slack rescaling method used for the loss function (slack rescaling or margin rescaling).
3. The heuristic used to select the gold derivation (maximum number of rules and minimum number of rules).

4. The trade-off between the training error and margin of the SVM’s linear kernel ($c$ parameter). For this parameter powers of 2 were used with exponents $\in [-9, 4]$ and step 1. The default value $c = 0.01$ was also used.

### 6.2 Evaluation Measures

To evaluate system performance for both our method and the baselines we have implemented two different metrics. Both evaluation metrics implemented were also used in (Quirk et al., 2015). An ideal system would output an AST that exactly matches the reference (correct) one. The first metric is inspired from this, and counts the number of correct predictions (accuracy). However, this metric is too strict and it gives no credit to predictions that are partially correct. For example, a prediction that has mispredicted only one terminal in target tree is considered as bad as an empty prediction.

The second metric aims to alleviate this and is a more forgiving measure that gives some credit to predictions that are partially correct. It calculates $F1$ (Equation 6.3) which is also known as harmonic mean of precision and recall (F-Measure with equally weighted precision and recall) on the set of productions. Precision (Equation 6.1) is defined as the number of predicted productions that appear in the correct derivation divided by the number of predicted productions. Recall (Equation 6.2) is defined as the number of predicted productions that appear in the correct derivation divided by the number of productions in the gold (correct) derivation.

\[
\text{Precision} = \frac{\# \text{Correct predicted productions}}{\# \text{Predicted productions}} \quad (6.1)
\]

\[
\text{Recall} = \frac{\# \text{Correct predicted productions}}{\# \text{Productions in gold derivation}} \quad (6.2)
\]

\[
F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6.3)
\]
6.3 Language to Code Results

6.3.1 Retrieval Baseline Results

In this section, the results for the character-based edit distance baseline are presented. The results for the token-based edit distance baseline are not presented since they were slightly worse (otherwise they were quite similar), thus no important inductions would be inferred by presenting them.

Tables 6.1 and 6.2 illustrate the performance of the character edit distance retrieval baseline for each fold for the Excel and the Geoquery dataset respectively. The mean and the variance across folds are also illustrated in the same tables.

The results indicate that this method fails in most of the cases to predict the correct program on both datasets. However, it always outputs a program, which is partially correct and usually has a few mistakes. The output programs of this method are quite similar to the correct ones. For this reason the method achieves high average F1 on the productions level for both datasets.

Most mistakes of the method are on the token level. For example, on the Excel dataset the method, might predict a Min function instead of a Max function, or do a filtering operation using the wrong column and value.

<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.74%</td>
<td>76.89</td>
</tr>
<tr>
<td>1</td>
<td>7.79%</td>
<td>77.72</td>
</tr>
<tr>
<td>2</td>
<td>8.03%</td>
<td>77.62</td>
</tr>
<tr>
<td>3</td>
<td>8.01%</td>
<td>76.83</td>
</tr>
<tr>
<td>4</td>
<td>7.58%</td>
<td>77.72</td>
</tr>
<tr>
<td>5</td>
<td>7.71%</td>
<td>75.61</td>
</tr>
<tr>
<td>6</td>
<td>8.11%</td>
<td>77.35</td>
</tr>
<tr>
<td>7</td>
<td>8.14%</td>
<td>78.65</td>
</tr>
<tr>
<td>8</td>
<td>7.71%</td>
<td>78.93</td>
</tr>
<tr>
<td>9</td>
<td>7.93%</td>
<td>76.08</td>
</tr>
</tbody>
</table>

Average 7.88% 77.34

Variance 0.0337 0.9654

Table 6.1: Experimental results for the character edit distance baseline on Excel.
Chapter 6. Experiments

<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.51%</td>
<td>74.93</td>
</tr>
<tr>
<td>1</td>
<td>2.77%</td>
<td>82.56</td>
</tr>
<tr>
<td>2</td>
<td>1.89%</td>
<td>79.22</td>
</tr>
<tr>
<td>3</td>
<td>3.14%</td>
<td>79.43</td>
</tr>
<tr>
<td>4</td>
<td>3.14%</td>
<td>78.93</td>
</tr>
<tr>
<td>5</td>
<td>2.39%</td>
<td>79.78</td>
</tr>
<tr>
<td>6</td>
<td>2.26%</td>
<td>78.50</td>
</tr>
<tr>
<td>7</td>
<td>2.77%</td>
<td>77.83</td>
</tr>
<tr>
<td>8</td>
<td>2.89%</td>
<td>82.79</td>
</tr>
<tr>
<td>9</td>
<td>1.64%</td>
<td>77.95</td>
</tr>
<tr>
<td>Average</td>
<td>2.44%</td>
<td>79.192</td>
</tr>
<tr>
<td>Variance</td>
<td>0.3242</td>
<td>4.68</td>
</tr>
</tbody>
</table>

Table 6.2: Experimental results for the character edit distance baseline on Geoquery.

6.3.2 Phrase-Based MT Results

This section presents the results of the phrase-based machine translation baseline. Tables 6.3 and 6.4 show the performance of the baseline for each fold for the Excel and the Geoquery dataset respectively along with the mean and the variance across folds.

On the Excel dataset this method always fails to predict the correct program, while on Geoquery the average accuracy across folds is only 5.18% and has a variance of 10.172. On the other hand, the method seems to achieve somewhat better results using the F1 on productions metric. However, these results are still quite bad and much lower than those for the retrieval baseline. The results indicate that this method fails in most of the cases to predict the correct program on both datasets. In many cases there is no valid program in the list of the top 100 predictions that the method outputs, which is one of the main reasons for the low F1 on productions. Lastly, this method might output programs that have invalid productions, something that cannot happen with the other methods, since the retrieval baseline outputs programs from the training set, and the tree transducer takes syntax into account and does not allow invalid productions.
6.3. Language to Code Results

<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0%</td>
<td>24.31</td>
</tr>
<tr>
<td>1</td>
<td>0.0%</td>
<td>38.52</td>
</tr>
<tr>
<td>2</td>
<td>0.0%</td>
<td>26.57</td>
</tr>
<tr>
<td>3</td>
<td>0.0%</td>
<td>31.07</td>
</tr>
<tr>
<td>4</td>
<td>0.0%</td>
<td>28.96</td>
</tr>
<tr>
<td>5</td>
<td>0.0%</td>
<td>29.05</td>
</tr>
<tr>
<td>6</td>
<td>0.0%</td>
<td>35.61</td>
</tr>
<tr>
<td>7</td>
<td>0.0%</td>
<td>26.80</td>
</tr>
<tr>
<td>8</td>
<td>0.0%</td>
<td>23.62</td>
</tr>
<tr>
<td>9</td>
<td>0.0%</td>
<td>28.34</td>
</tr>
<tr>
<td>Average</td>
<td>0.0%</td>
<td>29.285</td>
</tr>
<tr>
<td>Variance</td>
<td>0</td>
<td>19.992</td>
</tr>
</tbody>
</table>

**Table 6.3:** Experimental results for the phrase-based MT baseline on the Excel dataset.

6.3.3 Tree Transducer Results

The results of the tree transducer model described in this work, for each fold as well as the fold average and variance are presented in Tables 6.5 and 6.6. The results were obtained using the parameters selected by the tuning process on each dataset. For the Excel dataset the algorithm selected the following parameters:

- Loss function: Unordered CFG production errors.
- Slack rescaling method: Margin rescaling.
- Heuristic used to select the gold derivation: Maximum number of rules.
- Trade-off between the training error and margin of the SVM’s linear kernel: $2^0 = 1$.

While for the Geoquery one, it selected:

- Loss function: Unordered token precision * BP1 (see Appendix A).
- Slack rescaling method: Slack rescaling.
- Heuristic used to select the gold derivation: Minimum number of rules.
<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13.25%</td>
<td>26.50</td>
</tr>
<tr>
<td>1</td>
<td>4.82%</td>
<td>13.77</td>
</tr>
<tr>
<td>2</td>
<td>6.02%</td>
<td>23.65</td>
</tr>
<tr>
<td>3</td>
<td>4.82%</td>
<td>20.07</td>
</tr>
<tr>
<td>4</td>
<td>6.02%</td>
<td>17.99</td>
</tr>
<tr>
<td>5</td>
<td>0.0%</td>
<td>19.48</td>
</tr>
<tr>
<td>6</td>
<td>3.61%</td>
<td>10.20</td>
</tr>
<tr>
<td>7</td>
<td>3.61%</td>
<td>40.45</td>
</tr>
<tr>
<td>8</td>
<td>6.02%</td>
<td>19.21</td>
</tr>
<tr>
<td>9</td>
<td>3.61%</td>
<td>15.71</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>5.18%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.703</td>
</tr>
<tr>
<td></td>
<td>Variance</td>
<td>10.172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62.703</td>
</tr>
</tbody>
</table>

Table 6.4: Experimental results for the phrase-based MT baseline on the Geoquery dataset.

- Trade-off between the training error and margin of the SVM’s linear kernel:
  \[2^{-7} = 0.0078125.\]

The method achieves far better accuracy than both baselines in both datasets, with results on Geoquery being far better than those of the Excel dataset. One thing that must be noted is that there is a high variance among folds in Geoquery. A possible reason for this could be that some predicates are rare and in some folds they might appear only in the test set, thus being impossible for the system to translate them. Another possible reason could be the difference in the extracted grammars for each fold.

On the other hand, the F1 on productions, while much better than that of the phrase-based MT system, it is fairly lower than that of the retrieval baseline. The main reason for this is that in most of the mispredicted cases, the method fails to output a correct prediction, because it cannot cover the source tree of the test instance, which is trying to classify, due to a lack of appropriate rules in the grammar extracted from the training set. In all the cases that the method fails to cover the source tree, it outputs an empty prediction, while the retrieval baseline always outputs a prediction that is partially correct. These empty predictions, result in a significant decrease of F1. In order to make a more fair comparison and evaluate how good are the non-empty predictions of
6.3. Language to Code Results

<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
<th>F1 on Non-Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>37.11%</td>
<td>53.99</td>
<td>71.45</td>
</tr>
<tr>
<td>1</td>
<td>39.82%</td>
<td>54.70</td>
<td>73.72</td>
</tr>
<tr>
<td>2</td>
<td>36.66%</td>
<td>52.72</td>
<td>72.38</td>
</tr>
<tr>
<td>3</td>
<td>39.27%</td>
<td>53.72</td>
<td>72.39</td>
</tr>
<tr>
<td>4</td>
<td>36.66%</td>
<td>52.95</td>
<td>71.79</td>
</tr>
<tr>
<td>5</td>
<td>36.20%</td>
<td>51.24</td>
<td>70.78</td>
</tr>
<tr>
<td>6</td>
<td>39.82%</td>
<td>54.47</td>
<td>73.40</td>
</tr>
<tr>
<td>7</td>
<td>38.47%</td>
<td>53.26</td>
<td>71.78</td>
</tr>
<tr>
<td>8</td>
<td>37.11%</td>
<td>52.70</td>
<td>72.34</td>
</tr>
<tr>
<td>9</td>
<td>38.92%</td>
<td>54.13</td>
<td>73.85</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>38.00%</td>
<td>53.40</td>
</tr>
<tr>
<td></td>
<td>Variance</td>
<td>1.772</td>
<td>24.491</td>
</tr>
</tbody>
</table>

Table 6.5: Experimental results for the tree transducer method on the Excel dataset.

the method we also calculated the F1 on only the instances for which the method makes a non-empty prediction. The performance of the method measured by this modified F1 on productions metric is also illustrated in Tables 6.5 and 6.6. Taking this into account results in a significant increase of F1. The modified F1 for the Excel commands dataset is 72.48 and 94.153 for Geoquery, both with a small variance. This clearly suggests that while sometimes the method cannot output a prediction (low recall), when it manages to output one it is either a correct prediction or a very similar to the correct one (high precision).

Lastly, the method does not achieve results as good as those of state-of-the-art systems on Geoquery, which achieve almost 80% accuracy. On the Excel commands dataset, no statistical methods have been applied to compare to. However, the system described in (Gulwani and Marron, 2014) uses hand-crafted rules and also utilizes spreadsheet information achieves 94% accuracy on a subset of the data used in our experiments.

6.3.4 Results Comparison

This section summarizes the results of the previous sections. A comparison of the performance of all the methods, for both evaluation metrics is illustrated in Tables 6.7 and
<table>
<thead>
<tr>
<th>Fold</th>
<th>Accuracy</th>
<th>Productions F1</th>
<th>F1 on Non-Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>56.63%</td>
<td>59.93</td>
<td>90.44</td>
</tr>
<tr>
<td>1</td>
<td>63.86%</td>
<td>66.49</td>
<td>95.15</td>
</tr>
<tr>
<td>2</td>
<td>66.27%</td>
<td>68.24</td>
<td>96.01</td>
</tr>
<tr>
<td>3</td>
<td>68.68%</td>
<td>67.27</td>
<td>93.06</td>
</tr>
<tr>
<td>4</td>
<td>51.81%</td>
<td>56.15</td>
<td>93.22</td>
</tr>
<tr>
<td>5</td>
<td>65.07%</td>
<td>65.47</td>
<td>95.33</td>
</tr>
<tr>
<td>6</td>
<td>63.86%</td>
<td>62.53</td>
<td>92.69</td>
</tr>
<tr>
<td>7</td>
<td>59.04%</td>
<td>61.94</td>
<td>93.47</td>
</tr>
<tr>
<td>8</td>
<td>59.04%</td>
<td>60.40</td>
<td>96.41</td>
</tr>
<tr>
<td>9</td>
<td>61.45%</td>
<td>63.44</td>
<td>95.75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th></th>
<th>94.153</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>61.57%</td>
<td>63.186</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.785</td>
<td>12.827</td>
<td>3.193</td>
</tr>
</tbody>
</table>

**Table 6.6:** Experimental results for the tree transducer method on the Geoquery dataset.

<table>
<thead>
<tr>
<th>Method Used</th>
<th>Accuracy</th>
<th>StdErrAcc</th>
<th>Productions F1</th>
<th>StdErrF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Edit Distance</td>
<td>7.88%</td>
<td>0.06</td>
<td>77.34</td>
<td>0.31</td>
</tr>
<tr>
<td>Phrase-Based MT</td>
<td>0.0%</td>
<td>0</td>
<td>29.285</td>
<td>1.41</td>
</tr>
<tr>
<td>Tree Transducer</td>
<td>38.00%</td>
<td>0.42</td>
<td>53.40</td>
<td>1.56</td>
</tr>
</tbody>
</table>

**Table 6.7:** Comparison between the different methods on the Excel dataset.

6.8 for the Excel and the Geoquery dataset respectively. For each metric the standard error has also been estimated, which allows to measure if there is statistic significance when a method ihas better results than another. The formula for used to calculate the standard error (StdErr) is given in Equation 6.4. Once the standard error has been calculated, it is straightforward to create the error bars. For the top one, the standard error is added to the sample mean (average of the folds), while for the bottom one, the standard error is subtracted from the sample mean.

\[
StdErr = \frac{\sqrt{\text{variance}}}{\sqrt{\text{sample size}}} \tag{6.4}
\]

Finally, example output of the predictions of each method for the sentence “What are the highest points of all the states” can be viewed in Figure 6.1.
6.3. Language to Code Results

<table>
<thead>
<tr>
<th>Method Used</th>
<th>Accuracy</th>
<th>StdErrAcc</th>
<th>Productions</th>
<th>F1</th>
<th>StdErrF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Edit Distance</td>
<td>2.44%</td>
<td>0.18</td>
<td>79.192</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>Phrase-Based MT</td>
<td>5.18%</td>
<td>1.00</td>
<td>20.703</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td>Tree Transducer</td>
<td>61.57%</td>
<td>1.51</td>
<td>63.186</td>
<td>1.13</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8: Comparison between the different methods in the Geoquery dataset.

Figure 6.1: Example output for the various methods for the sentence “What are the highest points of all the states”. 
Chapter 7

Conclusion

7.1 Summary and Contribution of this Thesis

Semantic parsing is the task of transforming a natural language sentence into a computer executable complete MR, which is called a semantic parse, for a domain-specific application. Although this interesting problem has influenced a lot of works and has been studied extensively, it still remains an open problem. This work focused on semantic parsing of natural language into source code. Creating a semantic parse represented by source code would be of great benefit to inexperience users that do not know or like to program, as well as it would help to speed up software development. Additionally, performing the reverse task would have a significant impact in the code documentation process, as it could save a significant amount for developers.

The method employs a tree-to-tree transducer that translates the parse tree of the natural language into an abstract syntax tree of code. The methodology extracts an STSG grammar of translation rules from the train instances. The rules contained in this grammar either translate an elementary tree of the source language into an elementary tree of the target language or delete it. Then, the method learns to score possible derivations consisting of a series of grammar rules, by using a discriminative model that ranks the possible derivations using features for the rules of each derivation. The features of a derivations are the sum of the features of the rules contained in it. In order to make a prediction for a new test instance the method approximates the best scoring derivation using a chart-based dynamic programming algorithm, a beam-search and a cube-pruning heuristic.

To measure the performance of the method, a series of experiments was conducted on two different datasets. The first dataset consists of Excel commands along with
many possible descriptions in natural language for each one of the commands. The second one is based on queries, which can be executed by a database containing information about American geography, together with their description in natural language. The method was compared against two baselines methods introduced and used in (Quirk et al., 2015). The first baseline makes a predictions by predicting as output the meaning representation of the training instance of which the natural language description has the minimum edit distance from the test instance’s description (in natural language). The second one is a phrase-based machine translation system that is trained by linearising the tree pairs in order to produce a pair of series of tokens.

The main contributions of this work are the following:

1. It is the first method to perform tree-to-tree translation via a tree transducer. This contributes into ensuring proper syntax of the generated target language.

2. It uses a discriminative model that supports the definition of features on different levels (e.g., tokens, rules, derivation tree) instead of a generative model, which is usually employed in relevant work.

3. The methodology does not need to change in order to perform the reverse task with the exception that a ranked list of derivations should be returned, which only requires a simple modification of the decoding algorithm.

4. It is the first statistical method to perform experiments on the Excel commands dataset.

### 7.2 Future Work

In this final section possible extension for the method that would either be interesting to research, or would improve the method, and assist into achieving state-of-the-art results.

Measuring the impact of deletion rules (epsilon rules on the source side) when they are removed from the system would be a very interesting direction for future work. These rules were proposed by Cohn and Lapata (2007) to model the preservation of the most important parts of an elementary tree in the task of sentence compression. In this case, they model words that are not translated. However, it might be the case that this is not needed and only confuses the system.
The error analysis indicates that the main problem of the method is that the extracted grammar does not generalize well, because it does not have appropriate rules to cover the source trees of some test instances. To address this problem three main modifications are proposed. Firstly, different grammar extraction algorithms should be investigated. For instance, the current algorithm extracts the minimal set of rules for each pair of trees (source and target) separately. However, when the rules for an instance are extracted this process does not take into account the rest. It is very probable that a different set of rules could be extracted for that instance that covers more of the training data and thus might generalize better in new instances. Secondly, epsilon rules for the target side could be added to the grammar. This would be insertion rules that generate an elementary target tree from an empty tree. However, the system should have a way to limit the use of this in order to avoid overgeneration. Lastly, the experimental results show that while the retrieval baseline achieves terrible accuracy, it does return trees that are very similar to the reference one, since the F1 on productions is very high. From this we infer that it would be a reasonable idea to find the closest training instance and modify it in order to make a prediction for test instances that cannot be covered by the grammar.

Another interesting direction, would be to develop more features that could be used to score the possible derivations. Current features are only defined on the rule level (with the exception of the language model). However, other features such as those used in an MT decoder (e.g., word alignments) might be of great benefit to the model. Also, dataset specific features could have a great impact on system performance when combined with a better grammar. For example, in the Excel dataset a feature that checks if the column type used in the derivation matches that of the spreadsheet would help to avoid mispredicting the type of a column, which is a common mistake of the system.

Measuring the impact of the syntax parser used to parse the natural language sentences by creating and using gold parsers would be interesting as well. As an example, in the Excel dataset the parser fails to identify columns, when they are referenced.

The chart-based dynamic programming approximation algorithm uses only token features for pruning decisions. However, the algorithm could be extended to also prune on earlier parts of the derivation by using for example the log probability of the rules used in the derivation.

Additionally, we would like to perform experiments on the reverse task (natural language generation) for which suitable evaluation measures should also be developed.
Finally, we would like to also experiment on the newly released dataset of Quirk et al. (2015), which is much larger than the ones we experimented on. No experiments were performed on it due to time constraints, as it was released during writing this thesis.
Appendix A

T3 Loss Functions

Loss functions offered by T3:

1. Unordered token errors.
2. Unordered token errors with length penalty.
3. Unordered token precision.
4. (3) * brevity penalty (BP1) (Cohn and Lapata, 2007).
   \[ BP1 = \exp(1 - \max(1, \frac{c}{r})) \]
5. (3) * two-sided brevity penalty (BP2) (Cohn and Lapata, 2007).
   \[ BP2 = \exp(1 - \max(\frac{c}{r}, \frac{r}{c})) \]
6. Unordered n-gram errors (uniformly weighted).
7. Unordered n-gram precision (smoothed geometric mean).
8. (7) * BP1.
9. Unordered CFG production errors.
10. (9) * with length penalty.
11. Unordered CFG production precision.
12. Clipped unordered token errors.
15. Clipped unordered token F1.

16. Unordered token edit distance.

17. Asymmetric token hamming distance, similar to (2).
Bibliography


