Phrase-Based Statistical Translation of Programming Languages

Svetoslav Karaivanov
ETH Zurich
svskaraivanov@gmail.com

Veselin Raychev
ETH Zurich
veselin.raychev@inf.ethz.ch

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

Abstract

Phrase-based statistical machine translation approaches have been highly successful in translating between natural languages and are heavily used by commercial systems (e.g. Google Translate).

The main objective of this work is to investigate the applicability of these approaches for translating between *programming languages*. Towards that, we investigated several variants of the phrase-based translation approach: i) a direct application of the approach to programming languages, ii) a novel modification of the approach to incorporate the *grammatical structure* of the target programming language (so to avoid generating target programs which do not parse), and iii) a combination of ii) with custom rules added to improve the quality of the translation.

To experiment with the above systems, we investigated machine translation from C# to Java. For the training, which takes about 60 hours, we used a parallel corpus of $20,499$ C#-to-Java method translations. We then evaluated each of the three systems above by translating $1,000$ C# methods. Our experimental results indicate that with the most advanced system, about 60% of the translated methods compile (the top ranked) and out of a random sample of 50 correctly compiled methods, 68% (34 methods) were semantically equivalent to the reference solution.

1. Introduction

In this work, we investigate the application of phrase-based statistical machine translation to the problem of translating programs written in different programming languages and environments. The basic observation is that a large corpus of translated code is likely to contain regularities which can be effectively captured by a statistical learning approach.

Rule Based Approach and Combinations The alternative to statistical translation approaches is a rule-based approach, but as is well known in natural language translation [18], a full blown rule-based approach suffers from the fact that it is difficult to manually provide all of the possible rules and it can also be difficult to maintain (especially as programming platforms keep adding new APIs and features that need to be handled by the translation).

However, we believe that in the future, statistical translation systems will be combined with parts of traditional rule-based systems. In fact, in one of our configurations we experimented with a translation system that contains several user-defined rules. Other combinations are also possible: for instance, the phrase table which is automatically learned in the statistical approach can be consumed by a standard language translator.

Approximate Solutions To be effective, a statistical translation system need not always be exact – it has value in providing an approximate solution which is not semantic preserving yet is "close enough" to the desired solution. This is akin to recent statistical approaches for software synthesis [16, 17] where the synthesizer predicts (a set of) likely program completions or refactorings which are then shown to the programmer for further inspection.

This Work In this work, we first investigated a direct application of the phrase-based statistical translation approach to translating programs in different programming languages. While this approach has some success, it fails to take into account the fact that programming languages have an inherent structure captured by their grammar. This means that such an approach will invariably produce translations which are not grammatically correct.

To avoid this problem, we extended the phrase-based approach to take the programming language grammar into account and ensure that the translation only computes solutions which can be parsed by the language grammar. In statistical translation, the decoding (actual translation) process continually builds up *prefixes* until the entire source sentence is translated. Therefore, to take the grammar into account during translation, we need to continually ask the question of whether a given prefix can be *completed* in the given grammar. That is, even though the grammar cannot parse the prefix, the grammar may be able to parse some completion of that prefix. Then, if at any point, a completion of the prefix which the grammar can parse does not exist, we discard that prefix from further consideration.

To answer the question of whether a prefix can be extended without introducing new specialized parsing algorithms, we provide an optimized procedure that translates the original grammar into what we call a *prefix grammar*. Informally, the prefix grammar can parse all strings that can be parsed by the original grammar as well as their prefixes. Therefore, during machine translation, we can now directly check whether the current prefix can be parsed by the prefix grammar. Finally, to improve translation quality, we experimented by adding rules to the translation process.

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2661136.2661148

Example Translation: from C# to Java We experimented with our systems by translating programs written in C# to programs in the Java language. We trained the system on a parallel corpus of $20,499$ C#-to-Java method translations. We then evaluated the translation on $1,000$ C# methods. Our experimental results indicate that with our most advanced configuration, about 70% of the translated methods compile and out of a random sample of 50 correctly compiled methods, 68% (34 methods) were semantically equivalent to the reference solution while many of the remaining 16 were "almost equivalent" and could easily be fixed by a developer.

Main Contributions The main contributions of this paper are:

- An optimized procedure for translating a context-free grammar into a prefix grammar that is suitable for phrase-based statistical machine translation.
- An implementation which extends phrase-based translation to take language grammar into account.
- A detailed experimental evaluation indicating that with our most advanced configuration, about 60% of the translated methods compile and many of the translated methods are semantically equivalent to the reference solution.

We believe that this work is a promising step in automating the task of programming language translation and in understanding the pros and cons of adopting statistical approaches for this challenge.

Outline The paper proceeds by first reviewing the mathematical reasoning behind phrase-based translation. In then discusses the actual decoding (translation) process as well as how the prefix grammar is used. It follows with a discussion on how to obtain the prefix grammar and finally proceeds with a detailed implementation and evaluation of the approach.

2. The translation model

In this work, we adopt the phrase-based statistical translation model [8]. We next present a brief overview of this approach. The procedures of this model can be grouped into three main phases: training, translation and tuning.

2.1 Training phase

The training phase consists of several sub-phases to train the different components of a phrase-based machine translation: the phrase table and the language model. We next briefly describe the training process.

Step 1: Obtain Phrase table The process of constructing the phrase table is summarized in Fig. 1 and consists of three sub phases.

During the first sub phase "Parallel data collection", a set of input pairs $\langle m, n\rangle$ is obtained, where m and n denote the two methods (including the body of the method) that are translations of one another. For the purposes of translation, a method body (*e.g.,* m) is treated as a sentence made up of a sequence of tokens.

In the next sub phase "Word alignment", given the set of method pairs, an aligner discovers the most likely alignment between tokens of the methods of each pair. That is, for each method pair $\langle m, n\rangle$, the aligner produces a sequence containing the most likely alignment.

Finally, the phrase table is obtained. Each phrase consists of a sequence of tokens (of length more than one). A phrase table links a phrase from the source language to a phrase to the target language along with a score denoting the translation probability: the likelihood that the source phrase is a translation of the target phrase. Note that this translation (conditional) probability is in a sense "reversed": it denotes the probability that a phrase in the *target* language is translated to a phrase in the *source* language.

Step 2: Build a Statistical language model In addition to the phrase table, phrase-based machine translation requires a statistical model of the target language that scores correct and "fluent" sentences higher than incorrect ungrammatical or meaningless sentences. One goal of the phrase table is to help the translation engine stitch the phrases of the phrase table into meaningful programs.

A statistical language model assigns probabilities to sentences. If $w_1 w_2 ... w_l$ is a sentence where w_i is the i-th word, then its probability is computed as:

$$Pr(w_1 w_2 \dots w_l) = \sum_{i=1}^{l} Pr(w_i \mid w_1 \dots w_{i-1})$$

Since the probability of each word is computed as a probability on a potentially long prefix, for efficiency purposes, the product is often approximated. Typically, machine translation systems use an N-gram language model. This language model approximates a term $Pr(w_i \mid w_1 \dots w_{i-1})$ with the term $Pr(w_i \mid w_{i-N+1} \dots w_{i-1})$. For example, when $N = 3$, for each word w_i in the sentence, the approximation uses only the previous 2 words.

2.2 Translation phase

Given a phrase table and a statistical language model, one approach to find a translation $\hat{T}$ of a method S is via the Bayes rule:

$$\hat{T} = \underset{T}{\operatorname{argmax}}\, Pr(T \mid S) = \underset{T}{\operatorname{argmax}}\, \frac{Pr(S \mid T) Pr(T)}{Pr(S)} = \\ = \underset{T}{\operatorname{argmax}}\, Pr(S \mid T) Pr(T)$$

An important point here is that the target language model $Pr(T)$ can be trained on massive data purely for the *target* language, and is *independent* of the parallel corpus. Written with log-probabilities, the above equation becomes:

$$\hat{T} = \underset{T}{\operatorname{argmax}} \Big(\log\big(Pr(S \mid T)\big) + \log\big(Pr(T)\big) \Big)$$

However, in practice the above model does not take into account that our estimates of $Pr(S \mid T)$ and $Pr(T)$ may be biased due to the algorithms used in the phrase table and the language model. Thus, instead we use a more general model that allows assigning different weights to each of these components [13]. In this case:

$$\hat{T} = \underset{T}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i f_i(S, T)$$

where $f_i(S, T)$ are features on sentences and λ_i are weights. Two of the features we use are: i) the log-probabilities obtained from the phrase table, and ii) the statistical language model. This allows us to obtain the Bayes rule model as a special case of this more general model.

Subphase	Output	Example output
Parallel data collection	Method pairs	**C#**: `Console . WriteLine ( "Hello World!" ) ;` **Java**: `System . out . println ( "Hello World!" ) ;`
Word alignment	Aligned method pairs	**C#**: `Console . WriteLine ( "Hello World!" ) ;` **Java**: `System . out . println ( "Hello World!" ) ;`
Phrase table construction	Phrase table	**C# phrase** | **Java phrase** | **Score** i.e. Pr(**C# phrase** | **Java phrase**) `Console` | `System . out` | 0.8 `WriteLine` | `println` | 0.5 `.` | `.` | 0.7 `(` | `(` | 0.9 ... | ... | ...

Figure 1. Summary of the process of phrase table construction along with example input for translation between **C#** and **Java**.

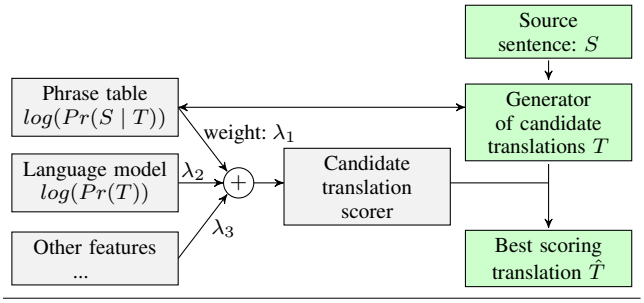


Figure 2. Architecture of the synthesis procedure in the decoding. This procedure generates a translation that maximizes a translation score, which is a linear combination of scores of phrase table, language model and other rules.

Given an input sentence S (in our case, a method body), the final translated sentence is obtained via a process shown in Fig. 2. In this approach, for a given source sentence S, the process first uses the phrase table to generate candidate translation sentences and these sentences are then scored with the equation above. Finally, the best scoring sentence $\hat{T}$ is returned as the translation.

2.3 Tuning phase

To optimize the quality of the translation system, one could optimize the weights λ_i. A common way to achieve this is by using a minimum-error rate training [12] on a tune dataset. This is an algorithm that optimizes the weights such that the quality of the translation on the tune dataset is as high as possible. To evaluate the quality of a translation, typically a BLEU [14] score is used.

2.4 Discussion of features

Since a weight-based system allows for more than just features on a phrase table and a language model, we give a brief overview of some of the features such systems use:

- $log\big(Pr(T)\big)$ - language model score.
- $log\big(Pr(S \mid T)\big)$ - phrase table translation probability.
- $log\big(Pr(T \mid S)\big)$ - phrase table reverse translation probability. Both phrase table probabilities are estimated by a maximum-likelihood estimation.
- A version of the phrase-table features with smoothed weights (e.g. with lexical weighting [7]).
- short/long phrase penalty - features that add penalty score when shorter (respectively longer) phrases from the phrase-table are used.

3. Decoding with Grammar

In this section we describe the decoding phase of the system. Given an input sentence, the decoding phase computes the most likely translations of that sentence. We describe a commonly used search algorithm [8] that generates and scores candidate translations as well as our modification to that algorithm to handle programming languages (essentially by leveraging the grammatical structure of the programming language). This algorithm approximates the best possible translation by gradually translating parts from the source to the target sentence employing a beam-search heuristic. We next describe the algorithm and show an example of its operation.

3.1 Definitions

Let us define a partial translation p of a sentence in the source language to be a pair $p = \langle S_c, L_t \rangle$. The intuition is that the *c* stands for *coverage* and the *t* stands for *translation*. The two components of the pair are:

- The set of (currently) translated words S_c: this set ranges over the indices of the words (tokens in our case) of the source sentence, that is, $S_c \subseteq \{1, 2, \ldots, m\}$ where m is the length of the source sentence (number of tokens) to be translated.
- The (currently) translated sequence L_t: this sequence ranges over the words (tokens in our case) of the target programming language.

We use $p.L_t$ to denote the translation of p and $p.S_c$ to denote the coverage of p. We define the length of a partial translation as $|p|$ to be $|p.L_t|$.

3.2 Algorithm Operation

Suppose that we wish to translate a sentence of length m and the maximum length of the resulting translation we can obtain is of length n. At the start of the decoding (translation) process, $n + 1$ stacks (the term stack was used in [8], but their implementation is not necessarily that of a stack), numbered as $St_0, St_1, \ldots St_n$ are initialized. Each stack St_k represents the set of partial translations which have length k. Every stack has a maximum capacity C (the number of partial translations it can hold).

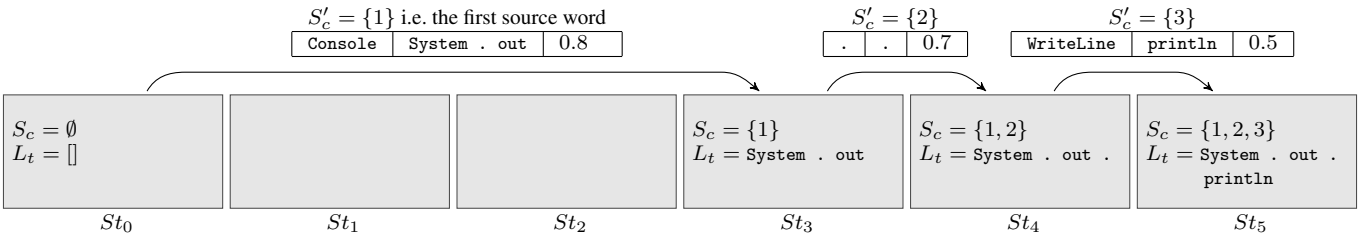


Figure 3. Example translation procedure of "`Console . WriteLine`" to "`System . out . println`". Each stack St_i holds the C best translations of length i. Each partial translation is scored (scores are not shown).

Algorithm 1 Beam-search expansion algorithm

Input: sentence s, maximum translation length n, phrase table, prefix grammar G
Output: stacks St_l ($l \in [0, n]$)

```
1: insert ⟨∅, []⟩ in St_0
2: for l ← 0 to n − 1 do
3:     for each partial translation p ∈ St_l do
4:         for each ph = w_{i+1} ... w_{i+k} ⊆ s do
5:             S'_c ← {i, i + 1, ..., i + k}
6:             if S'_c ∩ p.S_c = ∅ then
7:                 for ph' such that (ph, ph') ∈ phrase table do
8:                     L'_t ← p.L_t · ph'    // new partial translation
9:                     if L'_t ∉ L(G) then
10:                        continue    // check if in prefix grammar
11:                    end if
12:                    p' ← ⟨p.S_c ∪ S'_c, L'_t⟩
13:                    insert p' in St_{|p'|}
14:                    prune stack St_{|p'|} if necessary
                           (based on the translation score)
15:                end for
16:            end if
17:        end for
18:    end for
19: end for
```

We next explain the operation of the algorithm shown in Algorithm 1. For now, let us ignore Line 9 of the algorithm (this line captures our change and is discussed a little later). An illustration of the algorithm on an example is shown in Fig. 3. The process of generating translations is done by gradually building candidate prefixes, until all tokens from the source language are translated to the target language. The algorithm generates candidate translation prefixes of different length and then extends these (prefixed) translations.

At the beginning of the decoding phase, the empty partial translation $\langle \emptyset, [] \rangle$ is inserted in the stack St_0 (the translation for St_0 in Fig. 3). Then, at step $l \geq 0$, for each partial translation $p = \langle S_c, L_t \rangle \in St_l$, we extend p to obtain a longer partial translation as follows:

1. Pick a phrase $ph = w_i w_{i+1} \dots w_{i+k}$ (i.e. a sequence of tokens/words) from the source sentence that have not yet been translated. That is, for that phrase, its corresponding set of token indices S'_c satisfies the following constraints:

   - $S'_c = \{i, i + 1, i + k\} \subseteq \{1, \dots, m\}$, and
   - $S'_c \cap p.S_c = \emptyset$.

   In general, the phrase need *not* be a prefix (it is only required that it is a sequence of adjacent words). However, to illustrate the process, in our example in Fig. 3, we only include the case when we pick the first untranslated token from the source sentence. That is, initially, we pick the phrase $ph =$ `Console` with its corresponding $S'_c = \{1\}$, then we pick the phrase $ph =$ `.`, and so on.

2. We next translate all selected phrases from the previous step. That is, for the phrase ph, we take every translation ph' from the phrase table (i.e. the pair (ph, ph') is in the phrase table) and obtain new tokens L'_t for a partial translation by appending ph' to $p.L_t$. Therefore, for every phrase ph', we obtain:

$$L'_t \leftarrow p.L_t \cdot ph'$$

3. Finally, we update the respective stacks: we store the newly obtained partial translation $p' = \langle p.S_c \cup S'_c, L'_t \rangle$ in the stack $St_{|p'|}$. Note that the selected stack depends on the length of the produced translation, not the number of translated input tokens. Thus, in Fig. 3 the stacks St_1 and St_2 do not contain a partial translation as no translation is of length 1 or 2.

Scoring and Pruning Since the above procedure may exceed the capacity C of the stacks holding partial translations, we need to discard some of these translations. This is done by keeping scores of different features for each partial translation and only storing the C best scoring partial translations in each stack. For the example in Fig. 3, every stack contains up to one partial translation p.

Finally, we take the elements in each stack that are complete translations of the source sentence and return the best scoring one. For the example in Fig. 3, the last stack St_5 contains the only complete translation of input sentence, and hence, this is the final result of the translation for our example.

The above procedure generates the resulting translation by first building prefixes of it. At the same time, the model does not limit the order in which phrases from the source sentence are translated, which allows for phrase reordering between the source and the target sentence.

3.3 Our Modification: Including Language Grammar

The above algorithm does not take into account the fact that the programming language is defined by a formal grammar. To account for this fact, we slightly modify step 2 above. That is, at the end of step 2, we perform an additional check to see if the newly translated prefix sentence L'_t can actually be parsed by the language grammar (see Line 9 in Algorithm 1). If it can, L'_t is kept, otherwise, it is removed from further consideration. This prunes the search space that the machine translation engine would otherwise need to explore unnecessarily (as once L'_t cannot be parsed by the grammar of the language, any string built by appending to L'_t will also not be parse-able).

Key Question More precisely, we need to *efficiently* check more than whether the sentence L'_t can be parsed by the grammar of the language. It can easily be the case that L'_t cannot be parsed by the grammar, yet *there exists* a completion of L'_t that can be parsed and which the translation system would eventually find. Therefore, the question we aim to answer is:

Given a sequence of tokens (terminals) L'_t, is there an extension of L'_t which can be parsed by the grammar?

If the answer to this question is yes, then the machine translation system would keep L'_t, otherwise, it would discard L'_t. Further, it is important to answer this question as *efficiently* as possible as it is on the critical path of the decoding phase. To answer this question, in the next section we discuss a so-called prefix grammar. Essentially, given a grammar of a language, the prefix grammar can parse all prefixes of the strings that can be generated by the original grammar. It is this prefix grammar we discuss in the next section that we work with during the decoding phase.

4. Prefix Grammars

As we have seen in the previous section, during the decoding (translation) phase, at each step of the process we need to decide whether a given string L_t can be extended in a way that the extended string belongs to the grammar of the target language. To answer this question, we use a fairly standard procedure which instead of using new parsing algorithms, changes the grammar to what we refer to as a prefix grammar, in a way that enables us to reuse classic parsing algorithms and tools.

We do introduce few modifications to the procedure in order to speed-up the parsing query posed during the translation phase (our modifications produce about 3x speed-up for large sentences).

We next describe how to obtain such a prefix grammar from the original grammar.

Prefix Language Let L be a given language. We define $pre(L)$ to be the prefix language obtained from the language L, that is, $pre(L)$ contains all prefixes of any sentence in L. Formally:

$$w \in pre(L) \leftrightarrow \exists w' \in L.w \in pre(w')$$

where $pre(w)$ denotes the set of all prefixes for a word w.

Prefix grammar. The grammar which generates the language $pre(L)$ for a given language L is referred to as the *prefix grammar* of L. For instance, given a language of all syntactically correct Java programs and its grammar, we might be interested in obtaining the prefix grammar of that language, *i.e.,* the grammar which generates prefixes of syntactically correct Java programs.

Given a language L and its context-free grammar G (every such grammar follows the rules in Fig. 4), we next discuss a two-step approach to obtaining a prefix grammar G^p of $pre(L)$.

Step 1: Let S_T and S_{NT} be the sets of all terminals and (resp. non-terminals) in G. In this step, we go over every rule of the form $NT \rightarrow OE$ (where NT stands for non-terminal and OE stands for *orExpression*) and evaluate $pre(OE)$ as discussed below. The resulting set $pre(OE)$ will contain exactly the set of all prefixes of the language of OE.

⟨*rule*⟩ ::= ⟨*nonterminal*⟩ '->' ⟨*orExpression*⟩ ';'

⟨*orExpression*⟩ ::= ⟨*andExpression*⟩ ('|' ⟨*andExpression*⟩)*

⟨*andExpression*⟩ ::= (⟨*nonterminal*⟩ | ⟨*terminal*⟩ | ⟨*parenthesisedExpression*⟩ | ⟨*specialExpression*⟩)+

⟨*parenthesisedExpression*⟩ ::= '(' ⟨*orExpression*⟩ ')'

⟨*specialExpression*⟩ ::= (⟨*nonterminal*⟩ | ⟨*terminal*⟩ | ⟨*parenthesisedExpression*⟩) ('*' | '?' | '+')

Figure 4. The rules of a context-free grammar.

We use a helper function $\circ : W \times \mathcal{P}(W) \rightarrow \mathcal{P}(W)$ where W is a set of words (to help us define pre):

$$w \circ \{w_1, w_2, \ldots, w_k\} = \{ww_1, ww_2, \ldots, ww_k\}$$

We now recursively define the function pre as shown in Table 1. The domain of pre is the union of all possible *orExpression*s (OE), *andExpression*s (AE), *parenthesizedExpression*s and *specialExpression*s. The range is a set of words in the language used for describing the grammar rules of the original grammar with the slight modification that the set of nonterminal symbols S_{NT} is extended such that for every nonterminal symbol NT the symbol NT^{pre} is added. Intuitively, $pre(OE)$ is exactly the set of all prefixes of the language of OE. We note that it is the handling of the last three rules (e.g. for $E*, E+, E?$) which lets us obtain about 3x speed-ups over a naive unfolding.

Example For example $pre(A\ b\ C)$ (where A and C are nonterminal symbols and b is a terminal) can be calculated using the definition for $pre(A\ b\ C)$, expanding to $pre(A) \cup A \circ pre(b) \cup A\ b \circ pre(C)$, which in turn expands to $\{A^{pre}\} \cup \{A\ b\} \cup \{A\ b\ C^{pre}\}$.

Step 2: Once all $pre(OE)$ are computed, we next generate a set of new rules to be added to G^p (from the rules of G). In this step, each rule $NT \rightarrow OE$ in G is added to the new grammar G^p together with the rules:

$$pre(NT) \rightarrow \circ pre(OE)$$

Essentially, the above statement generates a set of rules from a single rule. The total number of newly generated rules to be added to G^p is $|pre(OE)|$.

Example Consider the grammar G:

$$G = \{S \rightarrow S_1 \mid S_2, S_1 \rightarrow aS_1b \mid ab, S_2 \rightarrow cS_2d \mid cd\}$$

The prefix grammar G^p then is:

$$G^p = G \ \cup \ \{\ pre(S \rightarrow S_1 \mid S_2)\ \ pre(S_1 \rightarrow aS_1b \mid ab)\ \ pre(S_2 \rightarrow cS_2d \mid cd)\ \}$$

where:

$$\begin{aligned} pre(S \rightarrow S_1 \mid S_2) &= S^{pre} \rightarrow \circ pre(S_1 \mid S_2) \\ &= S^{pre} \rightarrow \circ \{S_1^{pre}, S_2^{pre}\} \\ &= \{S^{pre} \rightarrow S_1^{pre}, S^{pre} \rightarrow S_2^{pre}\} \end{aligned}$$

$$\begin{aligned} pre(S_1 \rightarrow aS_1b \mid ab) &= \{S_1^{pre} \rightarrow a, S_1^{pre} \rightarrow aS_1^{pre}, \\ &\quad S_1^{pre} \rightarrow aS_1b, S_1^{pre} \rightarrow ab\ \} \end{aligned}$$

$$\begin{aligned} pre(S_2 \rightarrow cS_2d \mid cd) &= \{S_2^{pre} \rightarrow c, S_2^{pre} \rightarrow cS_2^{pre}, \\ &\quad S_2^{pre} \rightarrow cS_2d, S_2^{pre} \rightarrow cd\} \end{aligned}$$

Input	Output	Condition
NT	$\{NT^{pre}\}$	$NT \in S_{NT}$
T	$\{T\}$	$T \in S_T$
OE	$pre(AE_1) \cup pre(AE_2) \cup \cdots \cup pre(AE_k)$	$OE = AE_1 \mid AE_2 \mid \ldots \mid AE_k$ and every AE_i is a *andExpression*
AE	$pre(E_1) \cup E_1 \circ pre(E_2) \cup \ldots$ $\cup E_1\ E_2\ \ldots\ E_{k-1} \circ pre(E_k)$	$AE = E_1\ E_2\ \ldots\ E_k$ and each E_k is either *terminal*, *nonterminal* or *specialEepression*
$(\ OE\ )$	$pre(OE)$	OE is an *orExpression*
$E*$ $E+$ $E?$	$E* \circ pre(E)$ $E* \circ pre(E)$ $pre(E)$	E is a *terminal*, *nonterminal* or a *parenthesizedExpression*

Table 1. Definition of the function pre, expanding the grammar rules.

5. Implementation

We implemented a complete C# to Java statistical machine translation system based on several existing components:

- The Berkeley aligner [3] for sentence alignments.
- SRILM [19] for training language models, that is, the target language model $Pr(\text{Java})$.
- The Phrasal [4] translation engine.

We first collected parallel data to train the statistical system on, aligned the parallel data, built a phrase table with the Phrasal system, and modified the decoding phase of Phrasal to incorporate the C# and Java prefix grammars into the translation phase. Then, we tuned and evaluated the complete system. Finally, we experimented with manually introduced translation rules.

5.1 The prefix grammar implementation

We implemented a prefix version of the Java grammar by modifying the ANTLR Java grammar[1] according to the rules in Section 4. We also built a prefix C# grammar, which was a slightly more involved process. We describe the steps below:

The C# grammar We took the official *C# grammar v4 Language Specification* as basis and created a C# *ANTLR* grammar. However, several modification were needed to perform the translation to an ANTLR grammar. There are two main problems with the direct approach: the official specification contains a left recursion and C# by itself allows some keywords to be used also as identifiers (namely, the keywords found in LINQ) and it is not immediately obvious how this feature can be implemented in *ANTLR*. We solved the first problem via an application of *Paull's algorithm* [6] and the second by the trick mentioned in *The Definitive ANTLR 4 Reference* [15], p.209.

5.2 Parallel corpus mining tool

As one of the main purposes of the work is to automate the entire process and make it unsupervised, we developed a tool which automatically mines the translation pairs from a corpus of projects available in both C# and Java.

[1] https://github.com/antlr/grammars-v4/tree/master/java

General Problem with Mining Translations In general, we note that automatically (and even manually) obtaining a good parallel translation corpus between programming languages on already translated projects can be difficult. The reason is that a translation may break the method boundaries of the original source code and hence it may be difficult to figure out which statements in the source code are translated to which statements in the target. In general, we believe that a fruitful future direction to obtaining quality training data is to leverage Amazon Turk for language translation tasks, much in the way done for natural languages [9].

Implementation Nonetheless, we did implement a mining tool which uses both the Java and the C# *ANTLR* grammar to search for similar methods in a given project pair (C#/Java). The search is based on the premise that the enclosing classes of the methods in the translation pair share similar (almost equal) names. When such classes have been found, the tool continues to search for similar methods in the scope of the similar classes. We say that a pair of a C# method and a Java method is a *translation pair* when those two methods satisfy the following conditions:

- $n_1 \simeq n_2$
- $rtn_1 \simeq rtn_2$
- $np_1 = np_2$
- $ns_1 = ns_2$
- $SL_1 = SL_2$

where $n_i, rtn_i, np_i, ns_i, SL_i$ are respectively the i^{th} method's name, return type name, number of parameters, number of statements and set of all literals in the method body. Here $\simeq$ is defined as $w_1 \simeq w_2 \equiv lower(w_1) \sqsubseteq lower(w_2) \vee lower(w_2) \sqsubseteq lower(w_1)$, where $lower(w)$ denotes the lowercase version of the word w, and $a \sqsubseteq b$ means that a is a substring suffix of b. The equal sign has a default meaning.

Benchmarks We used the following projects available in both C# and Java to mine parallel data: *Db4o*, *Lucene*, *Hibernate*, *Quartz* and *Spring*. The number of mined pairs of translated methods for each open source project is listed in Table 2.

We divided the produced translation pairs into two disjoint buckets for training and testing with $20,499$ and 980 methods respectively. We also took 342 methods from the training bucket that we additionally put in a bucket for tuning parameters.

In Table 3, we give statistics for the number of tokens per sentence in the data we trained and evaluated on. Overall, most methods

Project	Number of mined translation pairs
Db4o	12419
Lucene	5721
Hibernate	2293
Quartz	698
Spring	518
Total	21649

Table 2. Number of matched methods extracted from open source projects available both in C# and Java.

Number of tokens per	C# method	Java method
Average	49.36	48.17
Minimum	7	6
Median	27	26
75%-ile	52	51
90%-ile	99.2	97
99%-ile	353	343
Maximum	7781	7783

Table 3. Statistics for the number of tokens in the training and testing corpus for C# and Java.

⟨*ruleSet*⟩ ::= ⟨*rule*⟩*

⟨*rule*⟩ ::= ⟨*tree*⟩ '`->`' ⟨*tree*⟩ '`;`'

⟨*tree*⟩ ::= '`(`' ⟨*non-terminal*⟩ (⟨*group*⟩ | ⟨*tree*⟩ | ⟨*terminal*⟩)* '`)`'

⟨*group*⟩ ::= '`[`' ⟨*non-terminal*⟩ '`]`' ⟨*number*⟩

Figure 5. Language to define custom translation rules.

consist of less than 100 tokens, and less than 50 tokens on average both for C# and Java. However, the data contains a heavy tail of overly long and complex methods.

5.3 Combining with a Rule-Based Approach

We expect a statistical machine translation system to be good in cases where phrases can capture the translation rules, for example API mapping. However, it may not perform as well for rules of arbitrary shape. To mitigate this limitation, we have extended our statistical machine translation to apply rules of more complicated shapes. The rules are currently manually defined by the user and we provide a mechanism to extend them.

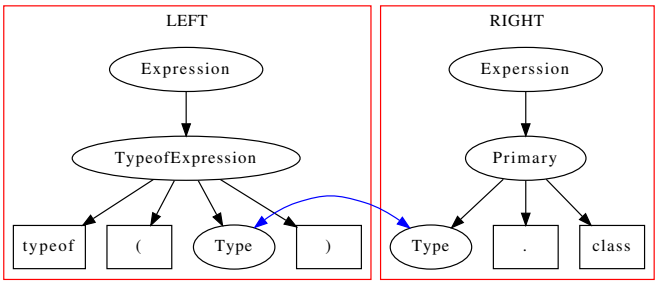
A custom rule is a pair of syntax trees (T_1, T_2) where T_1 is a syntax tree in the source language and T_2 is a syntax tree in the target language. The leaves of the trees can be both terminal and non-terminal symbols. We also define a mapping between the two sets of non-terminal symbols in the trees T_1 and T_2. We define the grammar for defining custom translation rules in Fig. 5.

We tested the effect of the rules experimentally by including a rule that coverts between `typeof` expression in C# and the dynamic type checking expression in Java as given in Fig. 6.

6. Evaluation

In our evaluation investigated the following key questions:

1. Can the SMT systems learn to synthesize translations that parse and compile?




```
(Expression (TypeofExpression 'typeof' '(' [Type]1 ')' ) )
                        →
(Expression (Primary [Type]1 '.' 'class' ) );
```

Figure 6. A rule that converts between the C# and the Java syntax of a dynamic type checking expression.

2. What is the effect of our improvements from Section 3.3 and Section 5.3?
3. Is SMT a feasible approach for translating between programs in different programming languages?

To answer the above questions, we trained three SMT systems from C# to Java on the mined parallel data from Section 5.2. All systems use the same phrase table and only differ in the way they generate translations. Next, we describe the SMT systems we evaluate on:

- *base* is our baseline system based on the standard features present in Phrasal like the ones listed in Section 2.4. This a typical system used in natural languages.
- *grammar* uses the same features as in *base*, but also uses the modified translation algorithm presented in Section 3.3 with a prefix grammar for Java.
- *combined* is based on our *grammar* system but augmented with a manually specified rule as described in Section 5.3.

We translated our evaluation data with each of the systems. Then, for every system we considered three evaluation metrics:

- BLEU score – a widely used metric in SMT for natural languages that scores the matching phrases between the generated and the reference translation. A score of 0% means no phrases matched, while a score of 100% means that all phrases matched.
- parse rate – the percentage of translated methods which parse with the Java grammar.
- compile rate – the percentage of translated methods which compiled.

We report each of the above rates on two translations. First, by only considering the best scoring translation for every method, and second – by taking the 30 best scoring translations and showing the one which results in the highest respective score. The results of our evaluation are summarized in Table 4.

BLEU score is not an indicator of quality First, we note that although our baseline system produces a good BLEU [14] score (natural language translation systems usually score far worse), its parse rate and compile rate are fairly low. In fact, although BLEU score is an important indicator for the quality of natural language translation systems, it does not reveal the quality of a programming language translation systems.

System	Description	BLEU [14]		Parse rate		Compile rate	
		first result	best of top 30	first result	best of top 30	first result	best of top 30
$base$	SMT baseline	86.2%	89.2%	57.7%	75.2%	48.5%	56.9%
$grammar$	SMT with prefix grammar	86.4%	89.7%	98.7%	99.2%	58.9%	67.9%
$combined$	SMT with prefix grammar and custom rules	86.7%	90.2%	98.7%	99.2%	60.7%	68.9%

Table 4. Evaluation results of the SMT systems from **C#** to **Java**.

Parse Rate In terms of parse rate, the baseline $base$ system produces translations that parse in less than 60% of the cases, and only increasing to around 75% when taking the best of the top 30 translations. This means that with the baseline system, for almost a quarter of the methods, none of the first 30 results parse. This result means that if we had implemented parsing only as a post-filtering step of the generated translations, we would have had problems generating meaningful translations.

In contrast, our second and third systems $grammar$ and $combined$, which incorporate grammars during translation, generate translated methods that parse for 98.7% of the cases. These systems failed to generate methods that parse for 0.8% of the cases – in these cases the SMT system did not generate any candidate translations likely due to sparseness of the phrase table.

Compile Rate The last two columns of Table 4 summarize the compile rate for each SMT system. Overall, adding grammars and rules increases the compile rate of the system with our best system having around 60% success rate.

6.1 Manual inspection

To get a better understanding of what a statistical machine translation is good and limited at, we inspected the translation results produced by our best system $combined$. We then randomly selected and manually inspected 50 translations that compile, and 50 translations that do not compile. For each translated method, we compared to the reference method in the original Java project and noted how the translation and the reference differ. The results for the two sets of methods are summarized in Table 5 and Table 6.

For 4 of the samples that compiled and 7 of the ones that did not we noticed that our algorithm for finding matching methods between C# and Java (Section 5.2) did not produce a pair of fully equivalent methods. The mismatches were similar (e.g. overloads) that were however not semantically equivalent. Thus, our translation was also not semantically equivalent to the reference.

Inspecting Successfully Compiled Methods From the set of translations that compiled, the vast majority of the translations were semantically equivalent to the reference translation. In 34 of the samples, the code is completely equivalent, while in 8 samples there is a difference in the exceptions that are declared in the `throws` section of the method. Java methods, as opposed to C#, must explicitly declare any exception they may potentially throw in their execution. Thus, when translating from C# to Java, the SMT system must guess or infer the set of exceptions. Our system made errors in both directions – for some methods the declaration in the translation was over-approximated with any `Exception`, while in others it was under-approximated with no `throws` declaration. Finally, 4 of the translations that compiled were semantically different and introduced an error due to the statistical machine translation. In two cases, the system changed a constant from an `enum`, in one it did not declare an inline class in a method, and in one case it incorrectly changed a method call to another method. However, even when the translation is not fully semantically equivalent, it is still useful if it is "close enough" to be manually adjusted.

Reason	Count
Fully equivalent to reference	**34**
Equivalent to reference except `throws` section	**8**
Semantically different	4
Reference mismatch	4
Total	**50**

Table 5. Results summary from manual inspection of 50 translations that compiled successfully.

Reason	Count
Wrong API	20
Coding convention (`TheMethod` vs `theMethod`)	17
Missing inline class	3
Wrong type	2
Incomplete	1
Reference mismatch	7
Total	**50**

Table 6. Results summary from manual inspection of 50 translations that did not compile.

Inspecting Methods which did not Compile From the set of translation that did not compile, we observed that 20 methods had API calls that the SMT system did not learn correctly. The translation included a call to a method (or object constructor) that did not exist. For the other 17 cases, the code did not compile, because there was a reference to a method in the project, but the names are slightly different between the C# and the Java versions of the project. In these cases, the reason for the compile failure was purely the difference in the coding conventions for method names – Java uses `lowerCamelCase`, while C# uses `UpperCamelCase`.

For the rest of the translations, the code did not compile due to differences in the features supported by Java and C#. For three of the samples, the translation did not include an inline class in the method like the reference solution, but instead tried to use a non-existing named class. This is because C# does not support anonymous classes like Java. For two other cases, the Java code needed to explicitly mention a type of an expression that the SMT system did not guess. Finally, for one example the SMT system did not manage to generate a complete translation.

6.2 Running times

Our current phrase-based implementation is based on research SMT systems that are not necessarily optimized for performance. The slowest of all steps was the word alignment step in the parallel data collection that took around 50 hours to align all of the $20,499$ sentence pairs in our training data (on an 64-bit Ubuntu machine with 2.13GHz Xeon E7-4830). The rest of the training steps took a few seconds only. All three systems we evaluated used the same training data. Additionally, the tuning step required 2 hours per system. Finally, the translation phase takes around 2 hours to generate the 30-best translations for the 980 methods in the evaluation data.

Summary Overall, we believe that statistical machine translation is an interesting approach to programming language translation and have shown that adding programming-language specific features improves the precision of the system. We believe that further research on the problem should focus on experimenting with statistical techniques combined with deeper semantic features.

7. Related Work

We next discuss several works most closely related to ours.

Programming languages and Statistical Methods The work most closely related to ours is Nguyen et al. [11] which uses a phrase-based SMT system to assist language migration. Their system reports encouraging BLEU scores, but suffers from high rates $(49.5\% - 58.6\%)$ of syntactically incorrect translations and does not employ language grammars. Further, their work does not provide detailed insights into the kinds of errors that arise as a result of translation (e.g. compilation and semantic errors).

Our recent work [17] discusses a new combination of statistical language models (N-gram, Recurrent Neural Networks) with static analysis (typestate and alias analysis). The aim of that work is to leverage statistical language models to perform code completion (within the same language), while this work focuses on a different programming challenge (and hence uses different statistical techniques). The work of Hindle et al. [5] uses an N-gram language model to predict the next program token.

Machine translation The last decades has seen tremendous improvements in the area of statistical machine translation systems. A good overview of such systems can be found in the book by Philipp Koehn [7]. Here, we briefly describe the work that is mostly related.

Och and Ney present a discriminative model for scoring translations based on features [13]. The work of Och further improves on the model by adding a minimum error rate training to train the features [12] and later Macherey et al. further improve it [10]. Gains in SMT translation quality came with the introduction of the BLEU score [14]. This score is of high importance to natural languages as a reasonably good predictor for translation quality, and a good optimization metric for a minimum error rate training. However, as we have seen, it may not be a good choice for programming language translation. Statistical machine translation has also been successfully applied to various application domains including semantic parsing of natural language queries [1] and opinion mining [2].

8. Conclusion and Future work

In this work we presented an approach to programming language translation based on statistical language models. The key idea of our approach is to integrate parsing queries to the programming language grammar into a phrase-based translation approach. We performed a detailed evaluation of several translation system variants instantiated to translate from C# to Java. The experimental results indicate that using our best system, roughly 60% of the resulting methods compiled. Further, manual inspection of 50 randomly selected methods which compiled indicated that 68% of the methods (34 methods) were semantically equivalent while many others were "close" to equivalent.

In the future, we plan to investigate combinations of statistical techniques with features that are more deeply aware of the program semantics, thus such an approach would be more reliant on advanced program analysis techniques. This will result in a translation system that is more accurate and effective for handling realistic programs.

References

[1] ANDREAS, J., VLACHOS, A., AND CLARK, S. Semantic parsing as machine translation. The Association for Computer Linguistics, pp. 47–52.

[2] BANEA, C., MIHALCEA, R., WIEBE, J., AND HASSAN, S. Multilingual subjectivity analysis using machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (Stroudsburg, PA, USA, 2008), EMNLP '08, Association for Computational Linguistics, pp. 127–135.

[3] Berkeley aligner. https://code.google.com/p/berkeleyaligner/.

[4] CER, D., GALLEY, M., JURAFSKY, D., AND MANNING, C. D. Phrasal: A statistical machine translation toolkit for exploring new model features. In *Proceedings of the NAACL HLT 2010 Demonstration Session* (Los Angeles, California, June 2010), Association for Computational Linguistics, pp. 9–12.

[5] HINDLE, A., BARR, E. T., SU, Z., GABEL, M., AND DEVANBU, P. On the naturalness of software. In *ICSE 2012* (2012).

[6] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[7] KOEHN, P. *Statistical Machine Translation*, 1st ed. Cambridge University Press, New York, NY, USA, 2010.

[8] KOEHN, P., OCH, F. J., AND MARCU, D. Statistical phrase-based translation. In *NAACL'2003 - Volume 1*.

[9] KUNCHUKUTTAN, A., ROY, S., PATEL, P., LADHA, K., GUPTA, S., KHAPRA, M. M., AND BHATTACHARYYA, P. Experiences in resource generation for machine translation through crowdsourcing. In *LREC* (2012), pp. 384–391.

[10] MACHEREY, W., OCH, F. J., THAYER, I., AND USZKOREIT, J. Lattice-based minimum error rate training for statistical machine translation. In *EMNLP* (2008), pp. 725–734.

[11] NGUYEN, A. T., NGUYEN, T. T., AND NGUYEN, T. N. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 651–654.

[12] OCH, F. J. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1* (Stroudsburg, PA, USA, 2003), ACL '03, Association for Computational Linguistics, pp. 160–167.

[13] OCH, F. J., AND NEY, H. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2002), ACL '02, Association for Computational Linguistics, pp. 295–302.

[14] PAPINENI, K., ROUKOS, S., WARD, T., AND ZHU, W.-J. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Stroudsburg, PA, USA, 2002), ACL '02, Association for Computational Linguistics, pp. 311–318.

[15] PARR, T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[16] RAYCHEV, V., SCHÄFER, M., SRIDHARAN, M., AND VECHEV, M. Refactoring with synthesis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2013), OOPSLA '13, ACM, pp. 339–354.

[17] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 419–428.

[18] SENELLART, J., DIENES, P., AND VRADI, T. New generation systran translation system. In *In Proceedings of MT Summit IIX Senellart J., Yang J., Rebollo A. 2003. SYSTRAN Intuitive Coding Technology. In Proceedings of MT Summit IX* (2001).

[19] STOLCKE, A. SRILM-an Extensible Language Modeling Toolkit. *International Conference on Spoken Language Processing* (2002).