FlashNormalize: Programming by Examples for Text Normalization

Dileep Kini∗
University of Illinois (UIUC)
Urbana, Illinois 61820

Sumit Gulwani
Microsoft Research
Redmond, Washington 98052

Abstract
Several applications including text-to-speech require some normalized format of non-standard words in various domains such as numbers, dates, and currencies and in various human languages. The traditional approach of manually constructing a program for such a normalization task requires expertise in both programming and target (human) language and further does not scale to a large number of domain, format, and target language combinations.

We propose to learn programs for such normalization tasks through examples. We present a domain-specific programming language that offers appropriate abstractions for succinctly describing such normalization tasks, and then present a novel search algorithm that can effectively learn programs in this language from input-output examples. We also briefly describe domain-specific heuristics for guiding users of our system to provide representative examples for normalization tasks related to that domain. Our experiments show that we are able to effectively learn desired programs for a variety of normalization tasks.

1 Introduction
Real world text contains words from various domains (like numbers, dates, currency amounts, email addresses, and phone numbers) in non-standard formats [Sproat, 2010]. It is desirable to “normalize” text by replacing such non-standard words (NSWs) with consistently formatted and contextually appropriate variants in several applications including machine translation, topic detection, text-to-speech (TTS) systems, training of automatic speech recognizers and spreadsheet functions.

The traditional technology for normalizing such NSWs involves manually writing down a normalization program, which often consists of a combination of ad hoc translation rules (e.g., for expanding dates or large numbers) along with some lookup tables (e.g., for month names or translation of single or double digit numbers). This process is not only time-consuming but also error prone since it involves manual programming and often requires to pair up a programmer with the target language expert. Most significantly, such a program is very specific and needs to be written for each domain (e.g., numbers) and the target language of normalization (e.g., English). To make matters worse, there can be multiple normalization formats, even for a given domain and a target language, each of which requires its own program. For instance, “1325” might be normalized into “one thousand three hundred twenty-five”, or may also be read as “one three two five” or “thirteen twenty-five” or “thirteen hundred and twenty five” depending on the context. In this paper, we propose automated learning of programs from examples for such tasks. This process is also referred to as inductive synthesis in some communities.

We propose a Programming-by-Examples technology called FlashNormalize for learning normalization programs. We identify an expressive domain-specific programming language (DSL) that offers appropriate abstractions for expressing such programs. This language describes the concept class. Our DSL is structured around four kinds of expressions: parse expressions that extract appropriate substrings from the input string, process expressions that transform the substrings using table lookups or functions provided by the language designer, concat expressions that concatenate various process expressions, and decision lists that allow for conditional behavior. The key technical contribution of the paper is a search algorithm for effectively learning programs in this DSL from input-output examples. Part of FlashNormalize’s search algorithm learns parse and process expressions (having learnt appropriate lookup tables) and builds over recent work on inductive synthesis by [Menon et al., 2013] that uses brute-force search to explore programs of increasing size as guided by an underlying DSL. While Menon’s approach works only on a single example, we show how to combine this approach with the idea of version-space algebras for inductive synthesis proposed by Lau et al. [Lau et al., 2000] to construct a hypothesis space of all programs that are consistent with all the user provided examples. In addition, we propose novel deductive top-down search algorithms to learn decision lists and concat expressions. Our algorithm for synthesizing decision lists falls into the paradigm of separate-and-conquer learning [Fünkranz, 1999]. In order to handle decision lists we identify a key

∗Work done by the author during internships at Microsoft.
Our key contributions are the following:

1. We have designed a rule-based DSL (Section 3) with support for lookup tables that is able to capture a range of Text-Normalization (TN) tasks. This DSL is our hypothesis space.

2. We describe learning algorithms for searching programs (within this hypothesis space) that are consistent with a set of input-output examples. In particular, we present a novel technique for learning decision lists (Section 4.1), and a novel method which combines bottom-up enumerative search (Section 4.2) with top-down deductive search (Section 4.3) for learning branches of the decision lists.

3. We describe strategies (Section 5.1) for helping users provide representative sets of examples and demonstrate their usefulness.

4. Using our techniques we show how FlashNormalize can learn programs for a range of TN tasks such as number names for 9 different languages, dates, phone numbers, time and measurements through examples (Section 5.2).

Related Work

Various language-specific techniques have also been proposed for French [Larreur et al., 1989], Russian [Sproat, 2010], Croatian [Beliga and Martinic-Ipsic, 2011]. Machine translation based methods have also been attempted [Schlippe et al., 2010] but these and other statistical translation methods require thousands of examples before they converge on approximately correct solutions. The upside to statistical methods is the ability to capture noisy input data that our methods do not possess. But FlashNormalize is much more useful when the user wants to provide a relatively small sample and can guarantee that it is consistent.

Our methodology of learning programs from examples falls into the broad effort of inductive synthesis. We point the user with intelligent inputs.

Comparison with FlashFill: From the technical view point the closest related work is FlashFill [Gulwani, 2011] which focuses on synthesizing spreadsheet programs. We treat text normalization as a string manipulation task, but a more sophisticated one than what has been attempted in FlashFill. Our task requires (a) learning larger programs in a more expressive DSL that is extensible and allows table lookups (b) dealing with more elaborate specification in the form of large number of examples; FlashFill handles tasks that require only a few examples. For (a) we bring forth two key innovations: (i) we combine a deductive (top-down) search with an enumerative (bottom-up) search to achieve an efficient synthesis algorithm. (ii) we present a generic technique for learning conditionals using maximal consistent covers — our technique scales to large number of examples unlike the greedy heuristic used in FlashFill. For (b) the user needs assistance in finding a representative set of examples for which we use two key strategies (i) modularity: wherein sub-procedures are learnt first and used as black-box functions for synthesizing higher level procedures, (ii) active learning: where we prompt the user with intelligent inputs.

2 Problem Motivation

We begin by describing three typical text normalization tasks and use them to motivate the problem we address in this paper.

Number Translations.

Translating sequence of digits into words representing the cardinal form is a scenario that arises in TTS for various languages. For example in English “72841” is spoken as “seventy two thousand eight hundred and forty one”, and in French “1473” is spoken as “mille quatre cent soixante-treize”. A TTS system engineer designing a system across various languages would need a different program for such translations for every single language.

Dates.

Consider the task of normalizing dates. The goal here is to transform a date written in ‘MMM dd, yyyy’ format to its expanded spoken form. Table 1 presents five representative examples for this scenario.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 08, 2065</td>
<td>January eighth twenty sixty five</td>
</tr>
<tr>
<td>Apr 23, 2006</td>
<td>April twenty third two thousand six</td>
</tr>
<tr>
<td>Oct 14, 2000</td>
<td>October fourteenth two thousand</td>
</tr>
<tr>
<td>Dec 31, 1804</td>
<td>December thirty first eighteen oh four</td>
</tr>
<tr>
<td>Aug 10, 1900</td>
<td>August tenth nineteen hundred</td>
</tr>
</tbody>
</table>

Table 1: Dates - MMM dd, yyyy.

What is common to all these examples is that we transform the MMM part into the expanded month and convert the dd part into its ordinal form. What is different in all of them is the way the years are written. In most cases (as in the first
example) the year yyyy is said in pairs, first two together and the last two together. But if the year is let’s say 2006 one obviously would not translate it to ‘twenty six’, but rather to ‘two thousand six’ or ‘two thousand and six’.

**Telephone numbers.**

Phone numbers are spoken differently across the globe. North America uses a system where a 10 digit number is spoken in three parts consisting of the area code (3 digits), exchange code (3 digits) and the subscriber number (4 digits). The area code might sometimes be omitted, and one might have optional country code at the beginning. The examples in Table 2 illustrates some of these challenges, along with the additional challenge of variation in the input formats.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4259037658</td>
<td>four two five / nine zero three / seven six five eight</td>
</tr>
<tr>
<td>(234) 7020671</td>
<td>two three four / seven zero two / zero six seven one</td>
</tr>
<tr>
<td>1 309 4442780</td>
<td>one / three zero nine / four four four</td>
</tr>
<tr>
<td>742-8537</td>
<td>seven four two / eight five three seven</td>
</tr>
</tbody>
</table>

Table 2: Examples for telephone number translation.

### 2.1 Problem Statement

We want to learn functions that take an input string and output a sequence of strings. The input in the first row in Table 1 is “jan 08, 2065” and the corresponding output is “January”, “eight”, “twenty”, “sixty”, “five”. A function $g$ is said to be **consistent** with a set of input-output examples $E$ if $g(\sigma) = \omega$ for each $(\sigma, \omega) \in E$. Our problem is the following: given a set of such input-output examples, **synthesize** a function that is consistent with all the examples.

### 3 Representation

We have identified a domain specific language (DSL) for representing functions in our concept space. The programs in this DSL are both (a) **expressive** enough to capture a range of normalization tasks and (b) **succinct** enough to be learnt efficiently.

FlashNormalize’s DSL consists of a **decision list** at the top, which is a chained sequence of if-then-else statements. Formally, it is an ordered sequence $d = (p_1, c_1), \ldots, (p_n, c_n)$ where each $p_i$ is a **Boolean predicate** and each $c_i$ is a concatenate expression with String $\rightarrow$ Bool and String $\rightarrow$ List(String) as their respective types. The final predicate $p_n$ is fixed to be true. For an input string $\sigma$, the value $d(\sigma)$ is defined as $c_i(\sigma)$ where $i$ is the least index such that $p_i(\sigma)$ evaluates to true. We use $D$ to denote the set of all decision lists.

A **concat expression** $c$ is an ordered sequence of process expressions $u_1, \ldots, u_n$. A concatenate expression $c$ applied to input $\sigma$ yields a list of strings $c(\sigma)$ obtained from concatenating the output values $u_i(\sigma)$ of process expressions in the order they appear in $c$. We use $C$ to denote the set of all concat expressions.

A **process expression** $u$ is a program that given an input string transforms it into a list of strings. It is allowed to be either a constant string (independent of the input) or a **Table lookup** applied to a parse expression. We use Table to indicate a finite sized map with signature String $\rightarrow$ List(String). Any such table can be thought of as a set of key/value pairs $(\kappa, \upsilon) \in String \times List(String)$ with no two of them having the same key. A program can use multiple tables (description of a program includes descriptions of the tables it uses).

In the examples in Figure 1 the substring ‘jan’ (“apr”) needs to converted to ‘January’ (“April”) or the string ‘8’ (“23”) should become ‘eighth’ (“twenty third”). Such transformations can be achieved by an appropriate table, like **month** and **cardinal** as used in Figure 1. Most often these tables represent core semantics relationship between substrings of the input and those of the output. These can either be specified by the user, or can even be learned by the system from sufficient examples.

**Parse expressions** are programs that extract substrings of the input. The space of possible parse expressions is described using a non-recursive grammar. An instance of such a grammar is presented in Listing 1. The reason for preferring a generic grammar to a fixed syntax is that algorithms designed for a grammar easily allow for future extensions without the need to modify the learning algorithm. Formally, a grammar is a 5-tuple $(S, \Phi, R, s, i)$ where:

(a) $S$ is a set of symbols denoting non-terminals, $^1$

(b) $\Phi$ is a set of functions, where each function $f$ has a specific signature $T_1, \ldots, T_k \rightarrow T$ where $T_1, \ldots, T_k$ are the types of the input and $T$ is the return type. The semantics of the functions is assumed to be given. A function can also have 0 arguments, in which case it is a constant.

(c) $R$ is a set of rules where each rule is either of the form $A := B$ or $A := f(B_0, \ldots, B_k)$ where $A, B, B_0, \ldots, B_k$ are symbols in $S$ and $f \in \Phi$

(d) a unique start symbol $s$ (in the sample below)

(e) a unique symbol in denoting the input variable ($\upsilon$ in the sample below). $R$ is not allowed to contain rules going out of this variable.

We explain some of the functions that constitute parse expressions. Consider the expression $Split(\upsilon, 0)$ used in $U_1$ in Figure 1, it is used to extract ‘jan’ from the input ‘jan 08, 2065’. The expression $Split(\sigma, i)$ returns the $i+1^{st}$ substring of $\sigma$ when split by the whitespace delimiter. The operation $Dig(\sigma, i)$ returns the $i+1^{st}$ substring of $\sigma$ composed entirely of digits, so $Dig(\upsilon, 0)$ yields ‘23’ on input ‘apr 23, 2006’ (used in Figure 1). The operator $Substr(\sigma, i, j)$ extracts the substring of length $j$ of $\sigma$ starting at index $i$. The index $i$ in all these operators is allowed to be negative which is a convention indicating that indexing is to be from right

$^1$Note that we do not have terminals in the grammar, instead parse trees terminate with functions of 0 arity.
to left. In case of \( \text{Substr}(\sigma, i, j) \) if the index \( i \) is negative then the substring extracted starts at index \( \ell(\sigma)+i-j+1 \) and ends at \( \ell(\sigma)+i \), where \( \ell(\sigma) \) is the length of \( \sigma \). The operator \text{Trim} removes all leading zeros from its argument. These and several other operators can be combined in different ways through a grammar. The domain expert is free to extend/limit the capabilities of the parse expressions appropriately for the task in question.

\[
\text{string } S := B \mid \text{SubStr}(B, k, k);
\]

\[
\text{string } B := v \mid \text{Split}(v, k) \mid \text{Dig}(v, k);
\]

\[
\text{int } k := -10 \mid -9 \mid .. \mid 10;
\]

\[
\text{string } v;
\]

Listing 1: Syntax of a subset of parse expressions presented as a grammar in Backus-Naur form. Each symbol is annotated with a type (eg: \text{string}) to denote the return type of the programs associated with it.

![Diagram](image)

Figure 1: Examples from Table 1 showing which substrings of the input are mapped to which substrings of the outputs using process and parse expressions.

<table>
<thead>
<tr>
<th>boolean formula</th>
<th>concatenate expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_3 != 0 )</td>
<td>( U_1, U_2, U_3, U_4 )</td>
</tr>
<tr>
<td>( y_2 = 0 \land y_4 != 0 )</td>
<td>( U_1, U_2, U_3, \text{\text{'thousand'}}, U_4 )</td>
</tr>
<tr>
<td>( y_2 = 0 )</td>
<td>( U_1, U_2, U_3, \text{\text{'thousand'}} )</td>
</tr>
<tr>
<td>( y_4 != 0 )</td>
<td>( U_1, U_2, U_3, \text{\text{'oh'}}, U_4 )</td>
</tr>
<tr>
<td>\text{true}</td>
<td>( U_1, U_2, U_3, \text{\text{'hundred'}} )</td>
</tr>
</tbody>
</table>

Table 3: decision list for the dates scenario. We use short hand \( y_i \) for the parse expression for extracting the \( i^{th} \) digit of yyyy part of the input. Process expressions \( U_i \) are the ones described in Figure 1.

A \textit{Boolean predicate} is a function of the type \text{String}→\text{Bool}, represented as conjunction of atomic predicates. Listing 2 is an example of atomic predicates described using a grammar. Each atomic predicate decides its truth value based on some feature of the input. For example, the function \text{CountSeq}(\sigma) counts the number of strings obtained by splitting \( \sigma \) with whitespace as delimiter, and the function \text{StrLen}(\sigma) counts the number of characters in \( \sigma \).

\[
\text{bool } A := \text{Equals}(N,k) \mid \text{Equals}(S,z);
\]

\[
\text{int } N := \text{CountSeq}(v) \mid \text{StrLen}(B);
\]

\[
\text{string } z := "0" \mid "1" \mid "00";
\]

Listing 2: Syntax of atomic predicates obtained by reusing rules in Listing 1

**Example 1** Table 3 provides an instance of a complete program in our DSL that is consistent with examples provided for Dates in Table 1.

## 4 Synthesis Algorithm

In this section we present our algorithm that takes as input a set of input-output example pairs and generates a program in the DSL such that the program is consistent with the examples. First, we define what it means for a set of examples to be consistent with respect to a class of functions.

**Definition 1** Given a class of functions \( \mathcal{F} \), a set of examples is said to be \( \mathcal{F} \)-consistent if there is some function \( g \in \mathcal{F} \) such that the examples are consistent with \( g \). Any such function \( g \) is called a witness for the set of examples.

Now, we describe our algorithm. It has two logically distinct phases performed in the following order:

1. A bottom-up learning of process expressions for individual examples.
2. A top-down search for decision lists and concat expressions that are consistent with all the examples.

A complete bottom-up approach is infeasible due to the fact that the number of different concat expressions and decision lists grow exponentially with their length. While designing a complete top-down approach requires knowing the semantics of the constructs and the shape of the grammar apriori, which we can afford for concat expressions and decision lists, but not for parse/process expressions whose structure is defined through a grammar. Hence, we perform a hybrid search as described above.

### 4.1 Learning decision lists

For a given set of examples we would like to search for the smallest decision list consistent with it, in accordance with Occam’s razor. Since this problem is computationally hard we propose a greedy heuristic that deduces a small decision list.

**Proposition 1** A decision list \((p_1, c_1), \ldots, (p_n, c_n)\) is consistent with examples \( E \) if there is a partition of \( E \) of the form \( E_1, \ldots, E_n \) such that \( E_i \) is the set of those examples in \( E \) whose input makes \( p_1, \ldots, p_{i-1} \) false and \( p_i \) true, and with each \( E_i \) being \( C \)-consistent (recall that \( C \) represents the class of concat expressions) with \( c_i \) as the witness.
First, we identify a key concept called \textit{maximal consistent cover} that enables us to find small partitions that are \textit{C}-consistent as above.

**Definition 2** Given a set of examples \(E\), a collection \(\{M_1, \ldots, M_k\} \subseteq 2^E\), is said to be the \textit{maximal \(F\)-consistent cover} of \(E\) if each \(M_i\) is \(F\)-consistent and for any \(M \subseteq E\), if \(M_i \subseteq M\) then \(M\) is not \(F\)-consistent.

Later on in this section we describe how to construct the maximal \(C\)-consistent cover with witnesses for its members. Now we show how to use this cover for constructing a small decision list. Observe that each part \(E_i\) in Proposition 1 has to be a subset of some member of the MCC cover. We design an iterative algorithm of the separate-and-conquer kind [Fürnkranz, 1999] to discover a small partition sequence \(E_1, \ldots, E_n\). After \(i\) iterations we would have produced a list \((p_1, c_1), \ldots, (p_i, c_i)\). The \(i\)th iteration eliminates examples \(E_i\). So, at the start of iteration \(i+1\) we would have left with examples \(R_{i+1} = E \setminus (E_1 \cup \ldots \cup E_i)\). Our goal in the \(i+1\)th iteration is to figure out a set \(E_{i+1} \subseteq R_{i+1}\) and a predicate \(p_{i+1}\) such that \(E_{i+1}\) is also a subset of some member of the MCC cover and \(p_{i+1}\) distinguishes \(E_{i+1}\) from \(R_{i+1}\). The concat expression \(c_{i+1}\) would be obtained from the witnesses of the MCC cover. In order to produce a small decision list we pick \(E_{i+1}\) as large as possible, while giving preference to members of the MCC cover over proper subsets of the members.

In Algorithm 1 we provide the pseudocode for the procedure \texttt{LearnProgram} which learns decision lists for a given set of examples. It iteratively learns the decision list by invoking \texttt{LearnBranch} on the remaining set of examples until every example is covered. (The notation \(E[p]\) is a shorthand for \(\{(\sigma, \omega) \in E \mid p(\sigma) = \text{true}\}\)). The function \texttt{LearnBranch} takes as input a set of examples and produces a pair \((p, c)\) of a predicate \(p\) and a concat expression \(c\). This pair is a choice for the first branch of a decision list that explains the example set. In lines 7 to 10 we compute a set of candidates of the form \(\langle p, M \rangle\) where \(M\) is a member of the MCC and \(p\) is a conjunctive predicate. This predicate computed by \texttt{LearnConj} is positive on all most examples in \(M\) and negative on all examples in \(R - M\). In lines 11 and 12 we try to pick that candidate \(\langle p, M \rangle\) whose predicate \(p\) can filter out \(M\) from the rest of the examples entirely, if not we pick a candidate whose predicate can filter out as many as possible. The chosen predicate \(p\) is returned along with the concat expression \(c(M)\) that witnesses \(M\) in the MCC.

The procedure \texttt{LearnConj} for computing conjunctive predicates is presented in Algorithm 2. It takes as input two sets of examples \(P\) and \(N\) and produces a set of predicates that are negative on all inputs in \(N\) and positive on as many as possible in \(P\). In line 1 it considers those atomic predicates that are positive on many examples in \(P\) with ties broken by looking at those that are positive on few examples in \(N\). If any of those atomic predicates already negate all examples in \(N\) then those are added to the result (line 4). For every other example \(a\) (line 6), it is combined with a new predicate (generated using a recursive call to \texttt{LearnConj}) that would remain positive on many examples in \(P\) that \(a\) is positive on and negative on all examples in \(N\) that \(a\) fails to be negative on.

### Algorithm 1: Learning decision list for set of examples.

```plaintext
function LearnProgram (E)
  let R ← E, d ← empty list;
  while R ≠ ∅ do
    let (p, c) ← LearnBranch (R);
    d ← d + (p, c);
    R ← R \ p;
  return d;
end function

function LearnBranch (R)
  Candidates ← ∅;
  foreach M ∈ MCC(R) do
    let p ∈ LearnConj (M, R - M) with max size(M[p]);
    Candidates ← Candidates ∪ \{ (p, M) \};
    if ∃ (p, M) ∈ Candidates with R[p] = M then
      return (p, c(M));
  let (p, M) ∈ Candidates with max size(R[p]);
  return (p, c(M));
end function
```

### Algorithm 2: Learning conjunctive predicates that pick most examples in \(P\) and discard all in \(N\).

```plaintext
function LearnConj (P, N)
  A ← top-k atoms sorted lexicographically by max size(P[a]) then by min size(N[a]);
  Result ← ∅;
  foreach a ∈ A do
    if N[a] = ∅ then
      Result ← Result ∪ \{ a \};
    else
      foreach p ∈ LearnConj (P[a], N[a]) do
        Result ← Result ∪ \{ a ∧ p \};
      return Result;
end function
```

### 4.2 Learning process expressions

We learn process expressions using a dynamic programming technique that builds up ideas of version space algebra by [Lau et al., 2000] and the idea of bottom-up program enumeration used in [Menon et al., 2013]. The technique we describe is applicable to any language of expressions described as a grammar and not limited to the one that we use in this paper.

**Version space** was first introduced by [Mitchell, 1982] to denote the set of all hypotheses (in a given hypothesis space) that are consistent with a given sample of labeled data. We use the term version space (VS) to describe a data structure that symbolically represents a partition of the programs. This data structure enjoys two properties:

1. It enables sorting a large number of programs into hierarchical groups with respect to their behavior on a given set of inputs.
2. It allows for an intersection procedure for producing a version space representing the intersection of the sets of
programs associated with different sets of inputs.

Given a non-recursive grammar we will show how to construct the VS data-structure for one input-output example. (Note that the tables and string constants can be treated as a part of the grammar because tables and examples are assumed to be given).

Formally a version space for a grammar \((S, \Phi, R, s, i_n)\) is a triple \((V, L, G)\), where \(V\) is a set of vertices, \(L : V \rightarrow S\) is a labelling function assigning a symbol to each vertex, and \(G \subseteq V \times \Phi^* \vee V \times V\) is a collection of edges. An edge is either a tuple \((u, f, v_1, \ldots, v_k)\) such that \(L(u) := f(L(v_1), \ldots, L(v_k)) \in R\) or a pair \((u, v)\) such that \(L(u) := L(v) \in R\). With every vertex \(u \in V\) we associate a set of programs \(\tilde{u}\), which is defined inductively as follows, for \(L(u) \neq \text{in}\):

\[
\tilde{u} = \{ f(p_1, \ldots, p_k) \mid \exists v_1, \ldots, v_k \text{ such that } \\
(u, f, v_1, \ldots, v_k) \in G \text{ and each } p_i \in \tilde{v}_i \} \\
\cup \{ t \mid (u, v) \in G \text{ and } t \notin \tilde{v} \}
\]

and if \(L(u) = \text{in}\), then \(\tilde{u} = \{ f_m \}\) where \(f_m\) is a function with no arguments that returns the input. This set is well defined because we deal with non-recursive grammars. Now we describe how to build such a data structure \(\text{VS}_u\), for a given input \(\sigma\). In \(\text{VS}_u\) each vertex \(v\) is associated with a value \(\text{val}(v)\) unique among all those \(u\) with \(L(v) = L(u)\), such that \(\tilde{v}\) represents those programs of \(L(v)\) that produce output \(\text{val}(v)\) when executed on input \(\sigma\). The way to achieve this is to add the edge \((u, f, v_1, \ldots, v_k)\) if and only if \(f\) evaluates to \(\text{val}(u)\) on inputs \((\text{val}(v_1), \ldots, \text{val}(v_k))\). This gives rise to a dynamic programming algorithm that computes this symbolic representation of sets of programs that produce the same output on the give inputs. We omit a detailed description of this algorithm as it can be derived from the inductive definition above.

Intersection of two version spaces on the same grammar is performed by taking a cross product, that is (i) for vertices \(u\) and \(v\) labelled by the same non-terminal we introduce new vertex \((u, v)\) in the intersection, and (ii) for edges \((u, f, v_1, \ldots, v_k)\) and \((u', f', v'_1, \ldots, v'_k)\) we add the edge \((u, u'), f(v'_1, v'_1), \ldots, (v_k, v'_k))\).

Given examples \((\sigma_1, \omega_1), \ldots, (\sigma_n, \omega_n)\) we consider the intersection of \(\text{VS}_{\omega_1}, \ldots, \text{VS}_{\omega_n}\) and check if it has a vertex \(v\) with \(L(v) = \omega_0\) and \(\text{val}(v) = (\omega_1, \ldots, \omega_n)\). Every program in \(\tilde{v}\) would be consistent with the given examples.

4.3 Learning concat expressions

Now, we describe an algorithm that learns the MCC for a given set of examples as required in learning decision lists. First we see how we synthesize a concat expression consistent with a single example \((\sigma, \omega)\). This amounts to searching for a sequence of process expressions that when applied to \(\sigma\) and concatenated yields \(\omega\). For any substring \(\omega'\) of \(\omega\), we can search for a process expression that produces \(\omega'\) by looking for a vertex \(v\) with \(L(v) = \omega_0\) in \(\text{VS}_u\), such that \(\text{val}(v) = \omega'\).

Next, we show how to search for a partition of the output string into substrings, such that for each substring there is a process expression that produces it on input \(\sigma\). Note that there are exponentially many different partitions of a list, but only \(\mathcal{O}(n^2)\) different substrings where \(n\) is the length of \(\omega\). Hence we can symbolically represent all partitions succinctly as follows: Given input-output pair \((\sigma, \omega)\), consider a directed acyclic graph \(G(\sigma, \omega) = (U, F, P)\) consisting of: vertices \(U = \{0, 1, \ldots, n\}\) where \(n = \text{len}(\omega)\), set of edges \(F \subseteq U \times U\) defined as

\[
F = \{ (i, j) \mid i < j, \exists v \in \text{VS}_u \text{ such that } \\
\tilde{v} \text{ is non-empty, } \text{val}(v) = \omega_{[i,j]} \}
\]

and an edge labelling function \(P : F \rightarrow V\) mapping edges to vertices in \(\text{VS}_\sigma\) where \(P(i, j)\) is the vertex that witnesses the inclusion of \((i, j)\) in \(F\). Observe that \(G(\sigma, \omega)\) is an acyclic graph in which a path from \(0\) to \(n\) denotes a concat expression that transforms \(\sigma\) to \(\omega\). So our problem of finding a concat expression reduces to searching for a path in this graph.

The next step is to use this idea to construct the MCC cover for a set of examples. We consider the directed graphs for each example and perform a parallel depth-first search. In every step of this search we pick an edge in each graph such that there is process expression common to all of them. When we cannot pick an edge for all examples we drop some and proceed along the rest in a greedy fashion. This search gives us concat expressions that explain subsets of examples. We maintain these subsets and use them to preemptively prune future attempts to find concat expression for subsets of examples that we have already found to be \(C\)-consistent.

Algorithm 3: Enumerating subsets of examples which are \(C\)-consistent.

```
f\(\text{function ConsistentSets} \{\{(\sigma_1, \omega_1, i_1), \ldots, (\sigma_n, \omega_n, i_n)\}\}\)
1    \[\text{if } \forall k : \text{len}(\omega_k) = i_k \text{ then}\]
2        \[\text{yield} \{ (\sigma_1, \omega_1), \ldots, (\sigma_n, \omega_n) \};\]
3    \[\text{else}\]
4        \[\text{foreach } S \subseteq \{1, \ldots, n\} \text{ do}\]
5            \[\text{if } \exists u \in S \exists j_s \in \text{VS}_u \text{ such that } \omega_s = (\omega_{i_s, j_s}) \text{ then}\]
6                \[\text{rec} \leftarrow \{ (\sigma, \omega, i_s, j_s) \mid s \in S \};\]
7                \[\text{foreach } r \in \text{ConsistentSets}(\text{rec}) \text{ do}\]
8                    \[\text{yield } r;\]
```

In Algorithm 3 we describe the pseudocode for enumerating subsets of examples that are consistent with a set of input-output examples. The function \(\text{ConsistentSets}\) takes as input a set of input-output examples each annotated with an index \(i\) indicating a position within the output \(\omega\). The first call to \(\text{ConsistentSets}\) is made with \(i = 0\). Lines 1 and 2, represent the base case in which the outputs of each example is covered. In lines 4 and 5 we enumerate those subsets of examples \(S\) for which a process expression \(u\) can be found that explains some prefix of each output \(\omega_s\) starting from the respective indices \(i_s\). In practice we do not go through all subsets but let the enumeration be guided by the process expressions that explain prefix of outputs of the examples. In lines 6 to 8 we do a recursive call that continues the search from the indices \(j_s\). In order to compute the MCC we just return the maximal subsets from all subsets obtained.
struct. In this subsection we see ways to tackle them which be large, or the tables might not be straightforward to con-

A key advantage of active learning is that one can learn required tables by asking the user for outputs on certain vari-
ations of the input. For example consider how 3 digit numbers are spoken in Portuguese: 101 becomes “cento e um”, 201 be-
comes “duzentos e um”, 301 becomes “trezentos e um” and so on. Therefore we require a table that maps 1, 2, 3 . . . to
cento, duzentos, trezentos and so on. This can be easily done by varying only the left most digit of the input and mapping the
input variation to the corresponding variation in the output. We use this strategy in number-translations to synthe-
size the required tables. Another way to help users figure out
counterexamples is through smart inputs. Smart inputs are a
maximal set of inputs such that any two of them can be distin-
guished by the boolean predicates allowed in the DSL. Such
inputs try to cover corner cases that might be missed other-
wise. We use these ideas to generate examples and tables in
an iterative fashion. In Figure 2 we describe how these are
put together to obtain a counterexample guided strategy in
synthesizing the correct program.

5 Implementation and Evaluation
In this section first we describe implementation strategies that
complement our synthesis algorithm, and then the experi-
mental evaluation we performed on real-world data.

5.1 Strategies
The learning algorithm described in Section 4 takes as input
a set of representative examples and descriptions of the re-
quired tables. In certain cases determining one/both of these
can be challenging. The required number of examples may
be large, or the tables might not be straightforward to con-
struct. In this subsection we see ways to tackle them which
will prove useful in the context of number translations.

Modularity is a software design principle that encourages
separation of a program into smaller pieces which can then be
reused. We employ this idea in synthesizing our programs.

Active Learning: Requiring the user to provide all repre-
sentative examples at the beginning can be too much to ask.
Active learning provides a setting which can guide the user
in finding the right examples. The DSL designer can encode
domain knowledge in the form a learner that suggests inputs
on which a synthesized program maybe wrong.

Traditionally, an active learner makes two kinds of queries,
(a) membership query, in which the learner presents an
input for which it would like to know the output (b) equiva-

5.2 Experiments
Now, we describe results showing the (a) expressiveness of
the programs in our DSL, (b) effectiveness of FlashNormal-
ize’s learning algorithms and strategies.

We first consider number translation scenarios as described
in Section 2 for 9 different languages. We assume that we are
given a table for translating 2 digit numbers, and learn n-digit
translators for n ranging from 3 to 6. We employ the idea of
modularity and synthesize n-digit translator using translator
for smaller lengths. For learning the correct translator it is
crucial that the user provides a representative set of exam-

...
Table 4: Experimental results for learning number-translators. Each language has four rows one for each translator (3 to 6, top to bottom). T and M denote number of test and membership queries made by the active learning method. E denotes number of examples used in synthesizing the program. tm denotes the time taken in seconds by the synthesis algorithm, and DI denotes the length of the decision list learned.

<table>
<thead>
<tr>
<th>Language</th>
<th>T</th>
<th>M</th>
<th>E</th>
<th>tm</th>
<th>DI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Russian</td>
<td>27</td>
<td>12</td>
<td>5</td>
<td>.13</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>17</td>
<td>8</td>
<td>.16</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>18</td>
<td>11</td>
<td>.23</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>183</td>
<td>14</td>
<td>17</td>
<td>.31</td>
<td>5</td>
</tr>
<tr>
<td>Polish</td>
<td>27</td>
<td>12</td>
<td>5</td>
<td>.15</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>15</td>
<td>8</td>
<td>.14</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>93</td>
<td>20</td>
<td>13</td>
<td>.20</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>210</td>
<td>34</td>
<td>27</td>
<td>.41</td>
<td>5</td>
</tr>
<tr>
<td>French</td>
<td>33</td>
<td>20</td>
<td>8</td>
<td>.12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>42</td>
<td>13</td>
<td>.16</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>142</td>
<td>57</td>
<td>34</td>
<td>.42</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>252</td>
<td>112</td>
<td>38</td>
<td>.77</td>
<td>10</td>
</tr>
<tr>
<td>Spanish</td>
<td>49</td>
<td>41</td>
<td>12</td>
<td>.14</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>44</td>
<td>14</td>
<td>.18</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>43</td>
<td>17</td>
<td>.26</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>72</td>
<td>42</td>
<td>1.6</td>
<td>11</td>
</tr>
<tr>
<td>English</td>
<td>20</td>
<td>4</td>
<td>4</td>
<td>.13</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>18</td>
<td>8</td>
<td>.14</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>19</td>
<td>10</td>
<td>.20</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>26</td>
<td>14</td>
<td>.26</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5: Experiments for learning other normalization tasks. Each language has four rows one for each translator (3 to 6, top to bottom). E, A, tm and DI denote the number of examples used in synthesizing the program, size of the data set, time take in seconds and the size of decision list learnt respectively.

<table>
<thead>
<tr>
<th>Task</th>
<th>E</th>
<th>A</th>
<th>tm</th>
<th>DI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dates</td>
<td>16</td>
<td>764</td>
<td>1.48</td>
<td>5</td>
</tr>
<tr>
<td>Measurements</td>
<td>15</td>
<td>820</td>
<td>1.46</td>
<td>7</td>
</tr>
<tr>
<td>Telephone</td>
<td>10</td>
<td>134</td>
<td>0.32</td>
<td>7</td>
</tr>
<tr>
<td>Time</td>
<td>63</td>
<td>966</td>
<td>13.7</td>
<td>15</td>
</tr>
</tbody>
</table>

6 Conclusion

In this paper, we have considered the problem of text normalization. Manually writing programs for such tasks is challenging as it requires both programming and domain expertise, and is complicated by the fact that this exercise would have to be repeated for every language and output format. We have proposed a Programming-by-Examples technology called FlashNormalize to automate this process in which the user only has to provide input-output examples. We show the effectiveness of our technique on real world problems. The core technical idea of the paper is the design of algorithms and strategies for learning programs in our DSL. While heuristic search, VSA methods and active learning have been studied in various communities we bring these complementary ideas together. We believe such a combination might be useful for scaling other synthesis tasks [Gulwani, 2010; 2012].

References