Compilation Error Repair: For the Student Programs, From the Student Programs

Anonymous Author(s)

ABSTRACT
Compile-time errors pose a major learning hurdle for students of introductory programming courses. Compiler error messages, while accurate, are targeted at seasoned programmers and seem cryptic and distracting to beginners. The problem of pedagogically-inspired program repair that can provide more helpful feedback on erroneous programs has generated a lot of interest in recent years. In this work, we report TRACER (Targeted RepAir of Compilation ER-
ors), an end-to-end system for performing repairs on compilation errors, aimed at introductory programmers.

TRACER invokes a novel combination of tools from programming language theory and deep learning, and offers far more relevant repairs than existing works. TRACER distinguishes itself in generating repairs that not only enable successful compilation of erroneous programs, but those that are very close to repairs actually performed by students on similar errors. This is in sharp contrast to the relatively meek evaluation criteria used by existing works, such as compilation success rates. These targeted corrections are much more useful in pedagogical situations than code that merely compiles. TRACER learns repairs from erroneous student programs themselves and requires minimal expert intervention. It scales well and can be used to offer real-time feedback to students during a programming lab or tutorial session.

We report experimental results of TRACER on 4500 erroneous C programs written by students in a freshman year programming course. In 68% of these, TRACER recommends a repair that exactly matches the repair expected by the student. In 79.27% of these programs, TRACER produces a compilable repair. On a further set of 3200 programs that require errors to be fixed on multiple lines, TRACER enjoys a success rate of 44% compared to the 27% success rate offered by the state-of-the-art technique DeepFix.

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE-SEET 18, May 27-June 3 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5660-2/18/05...
https://doi.org/10.1145/3183377.3183383

1 INTRODUCTION
The field of automated code repair has traditionally focused on fixing logical errors, upon being given failing test-cases or templates, while not paying much attention to compilation errors [15]. This was deemed acceptable, given advancements in compiler techniques which made identifying and fixing compile-time errors by experienced developers, relatively straightforward.

However, as noted by Traver [25], compiler error messages are cryptic, and often an impediment to effective programming for an average programmer. Unfortunately, this issue has been largely ignored by compiler designers and better error messages are usually a low priority feature [25]. For those beginning to learn a new language, and hence unfamiliar with programming constructs of that language, compiler errors can be very confusing and time consuming to fix [5]. This is especially true for those who lack prior experience in any form of programming.

Figures 1 and 2 illustrate actual attempts by students in the very first lab session of an introductory course on C programming, as well as the actual fixes proposed by our method TRACER. The captions list the messages returned by the Clang compiler [11], a popular compiler for the C language. It is clear that the compiler error messages in these examples are not very informative for a student who has just been introduced to the concept of formal programming with datatypes and operators.

```
1 #include<stdio.h>
2 int main(){
3     int a;
4     scanf("%d", &a);
5     printf("ans=%d",
6             a+10);
7     return 0;
8 }
```

```Figure 1: Left: erroneous program, Right: fix by TRACER. The compiler message read: Line-4, Column-9: warning: format ”%d” expects argument of type ‘int ’, but argument 2 has type ‘int’.
```

```
1 #include<stdio.h>
2 int main(){
3     int x,x1,d;
4     // ...
5     d=(x=x1)(x=x1);
6     return d;
7 }
```

```Figure 2: Left: erroneous program, Right: fix by TRACER. The compiler message read: Line-5, Column-11: error: called object type ‘int ’ is not a function or function pointer.
```
In the first case, the error message does not provide any valuable feedback to a student unfamiliar with the concept of pass-by-reference vs pass-by-value. It may be frustrating to the student that a format that is valid for printf (%d, followed by variable name) is being flagged as an error for the scanf invocation. In the second case, the compiler error message may actually mislead the student since the error in the program is simply the omission of an arithmetic operator whereas the compiler is interpreting it as an illegal function invocation, simply because parentheses are involved.

These gaps arise since compilers messages are written for expert programmers and assume that the programmer has a thorough understanding of advanced concepts such as variable addresses, pointers and function invocation. However, these concepts are generally covered much later in a programming course. Thus, a novice programmer is unable to comprehend the error message or the cause of the error. In our dataset, around half of the students made similar errors and although the fix is simple in all these cases, students did require the help of a teaching assistant to understand the error messages and apply the fixes.

Although it is true that the time taken by students to fix common compile-time errors decreases over time as they get more comfortable with grammar of the language and adapt to the compiler messages, in a typical course, it takes a considerable amount of effort and supervision from teaching assistants who have to not only help students correct mistakes, but also help them understand the cause of the error.

With a move towards Massive Open Online Courses (MOOCs) where thousands of students may enroll, it is infeasible to provide human assistance in this manner, even in the initial phase. Moreover, there is evidence [5, 6] that across such programming courses, the errors made by students in the initial phases are quite similar. This points to a lucrative potential for automating this repetitive and monotonous role played by teaching assistants.

The work of Traver [25] attempts to address this problem by offering more informative error messages that would aid programmers in easier diagnoses. There has also been work on designing custom compilers for novice users [24]. These compilers aim for better error recovery and correction than standard compilers, so as to offer better feedback. However, these require a significant amount of effort from compiler designers, and the effort has to be replicated for every compiler being used. Moreover, studies [16] have shown that additional information in the form of enhanced error messages does not seem to be very helpful, especially for novice programmers.

### 1.1 Our Contributions

In this work we develop TRACER (Targeted RepAir of Compilation ERrors), a method for automatically generating template repairs for buggy programs that face compilation errors. TRACER is based on a realization that the goal of program repair in a pedagogical setting is not to simply eliminate compilation errors using any means possible, but rather to reveal the omissions to the students, so that they may learn how to correct similar errors themselves in the future [27]. In fact, it may be argued that offering the repaired program to the student directly which many existing works do [2, 6, 7], defeats the purpose of learning and may even pose challenges in course evaluations.

The techniques adopted by TRACER for program repair are motivated by a common observation [6, 7] that in practice, programmers (especially novice programmers) use a rather small subset of rules and productions of the entire grammar. Thus, it should be possible to repair an erroneous program by mapping it to a similar program (from past code obtained from students themselves) that is known to be compilable, hence (at least) syntactically correct.

This approach of automatically predicting repairs for a non-compiling program has gained interest recently. However, the guarantees offered by existing works [2, 6, 20] are rather modest. The repair mechanisms therein simply claim to produce compilable code in a large fraction of instances. However, this in itself does not solve the pedagogical problem where the goal is to educate the student, and not merely remove/reduce compilation errors.

The point becomes even more significant given the fact that trivial fixes such as deleting the error line, or replacing it with a standard statement can make the compiler error disappear, while not achieving the fix desired by the student. To ameliorate the problem, TRACER offers targeted corrections that pinpoint the source of the error, as well as recommend the fix actually desired by the student, thereby offering compilable code as a by-product. Figures 1 and 2 demonstrate some of the fixes TRACER can successfully apply to actual programs.

To this end, TRACER adopts a modular, three-phase methodology for code repair which involves:

1. **Error Localization**: TRACER locates the line(s) where repair must be performed.
2. **Abstract Code Repair**: TRACER attempts to identify the intent of student and recommends an abstracted form of the repair to the erroneous line(s), based on sequence-to-sequence prediction techniques that use recurrent neural networks.
3. **Concretization**: TRACER finally converts the abstract repair into actual code that can be compiled.

The above presents a significant departure from existing works [2, 6, 20], which also adopt deep learning techniques such as recurrent neural networks, but in a very monolithic manner. It is common in existing works to simply feed the entire erroneous program into a deep network and expect repairs as an output. In contrast, TRACER’s modular approach to error repair offers several key advantages over the state-of-the-art.

1. To the best of our knowledge, TRACER is the first system to be able to successfully reproduce the exact fix expected by the student on an erroneous program, and not merely reduce compilation errors.
2. Even when comparing compilation rates, TRACER offers success rates that are far superior to the state-of-the-art.
3. TRACER offers repairs in both abstract and concretized forms. Depending on the learning objectives set by the instructor, either may be offered to students.
4. In particular, the concretized code may be redacted and just the abstract form offered if it is desired that the student identify the form of the error from this feedback rather than...
Table 1: Examples of source-target pairs for single-line errors. Each row describes an actual error case found in our data. The first column describes the (erroneous) source line and its abstraction. The second column describes the target line (extracted from repairs attempted by the student) and its abstraction. Finally, the third column describes the abstract and concrete versions of the top-ranked repair suggested by TRACER. TRACER produced compilable code in all these cases. More importantly in most cases, TRACER is able to predict the exact fix expected by the student. TRACER is robust to even inappropriate (albeit compilable) fixes made by students. For instance, in the example in row 3, the target (student) code will compile with warnings but most likely face runtime errors, but TRACER’s top suggestion is the most appropriate fix for the corresponding error.

<table>
<thead>
<tr>
<th>#</th>
<th>Source-line and Source-abstraction</th>
<th>Target-line and Target-abstraction</th>
<th>TRACER’s Top Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(d = (x \times 1)(x \times 1);)</td>
<td>(d = (x \times 1)(x \times 1);)</td>
<td>(d = (x \times 1)(x \times 1);)</td>
</tr>
<tr>
<td>2</td>
<td>(p + a' = p;) (CH ‘LITERAL_C' = CH;) (p = p + a';)</td>
<td>(p = p + a';) (CH = CH + 'LITERAL_C';) (p = p + a';)</td>
<td>(p = p + a';) (CH = CH + 'LITERAL_C';)</td>
</tr>
<tr>
<td>3</td>
<td>(printf(c);) (printf(c);) (printf(c);)</td>
<td>(printf(c);) (printf(c);) (printf(c);)</td>
<td>(printf(c);) (printf(c);) (printf(c);)</td>
</tr>
<tr>
<td>4</td>
<td>(if[a][i==a][j &amp; i]!=j])(if[a][i==a][j &amp; i]!=j]) (if[a][i==a][j &amp; i]!=j])</td>
<td>(if[a][i==a][j &amp; i]!=j])(if[a][i==a][j &amp; i]!=j]) (if[a][i==a][j &amp; i]!=j])</td>
<td>(if[a][i==a][j &amp; i]!=j])(if[a][i==a][j &amp; i]!=j]) (if[a][i==a][j &amp; i]!=j])</td>
</tr>
<tr>
<td>5</td>
<td>(printf(%d, a + b/0);) (printf(%d, a + b/0);)</td>
<td>(printf(%d, a + b/0);) (printf(%d, a + b/0);)</td>
<td>(printf(%d, a + b/0);) (printf(%d, a + b/0);)</td>
</tr>
<tr>
<td>6</td>
<td>{ while (a &gt; 0) } { while (a &gt; 0) } { while (a &gt; 0) }</td>
<td>{ while (a &gt; 0) } { while (a &gt; 0) } { while (a &gt; 0) }</td>
<td>{ while (a &gt; 0) } { while (a &gt; 0) } { while (a &gt; 0) }</td>
</tr>
</tbody>
</table>

simply receive corrected code. This is not possible with existing techniques.

The modular structure of TRACER also helps it harness the power of deep learning techniques in a focused manner. In particular, the abstraction phase implicitly performs a vocabulary compression step that greatly eases the working of neural networks.

We note that although we present the TRACER system for a C-programming environment, versions of TRACER may be readily developed for other languages as well. As we shall see, the language specific components of the system are minimal, as is the manual effort required to port the system to a new programming language.

### 1.2 Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we describe the procedures followed to curate and prepare training data for training the TRACER system. In Section 3 we present details of all three modules that constitute the TRACER system. In Section 4 we present a brief overview of the deep learning techniques used by TRACER for sake of completeness. In Section 5 we present details of an extensive evaluation of the TRACER system on programs taken from an actual introductory programming course. In Section 6 we present a more detailed literature review. Sections 7 and 8 then respectively, outline some future directions for this effort, and conclude the discussion.

### 2 DATA PREPARATION

Since TRACER learns to recommend error repairs from student programs themselves, we obtained student submissions in various lab assignments from the 2015-2016 fall semester offering of Introductory C Programming course at the authors’ university. Below we describe the steps taken to create a training subset for TRACER from these submissions.

#### 2.1 Raw Data Collection

Our data was collected using an online system that captures code submissions by students while they attempt their weekly lab assignments. The assignments are attempted under the invigilation of teaching assistants. The system is capable of taking snapshots of the student code at every compilation request, as well as at regular intervals by default. Thus, for the same student attempting a given question, the system gives us a sequence of programs that track the progress of that student in solving that question.

#### 2.2 Source-Target Pair Identification

Given this raw data, we filtered all codes to obtain source-target program pairs as follows. We identified successive snapshots of the student code (say \(C_t\) and \(C_{t+1}\) such that

1. Compilation of \(C_t\) resulted in at least one compilation error
2. Compilation of \(C_{t+1}\) did not produce any compilation errors

\(C_t\) is called the Source Program and \(C_{t+1}\) is called the Target Program. These program pairs were further filtered out to those pairs where the programs \(C_t\) and \(C_{t+1}\) differ at a single line (single-line edits) and those that differed on multiple lines (multi-line edits).

Ostensibly, the former correspond to programs where the error in the program was confined to a single line whereas in the latter, the error, and hence the student edits, were present on multiple lines. For single-line edit program pairs, the line in \(C_t\) that was changed is called the Source Line whereas the same line in \(C_{t+1}\) is called the Target Line.

For example, in Figures 1 (respectively 2), line number 4 (respectively line number 5) in the left and the right hand programs is the source and the target line for that pair of programs. We only used single-line edit program pairs for training TRACER. However, we do report results on multi-line edit program pairs as well (see Section 3.5).
We refer the reader to Figure 3 for an overview of our dataset and also an initial look at the performance of TRACER on this dataset. Table 3a lists all types of compilation errors that were present in at least 50 submissions in our dataset. Figure-3b then charts the frequency of occurrence of all these error types in our training and test sets.

Recall that even though we consider only single-line edit source-target pairs for training, our dataset still contains source programs with multiple compilation errors. Also notice that TRACER does well not only on frequent error types such as missing identifiers (E1), but also very rare ones such as invalid digit in constant (E12).

### 3 TRACER: AN END-TO-END SYSTEM FOR TARGETED REPAIR OF COMPILATION ERRORS

In this section, we describe the technical details of the TRACER system. For most of this discussion, we will focus only on program pairs that have single-line edits. We will pay special attention to multi-line edit programs in Section 3.5. Recall that for single-line edit program pairs, we can identify a source and a target line, the target line being the fix the student applied to rectify the error in the source line.

TRACER treats compilation-error repair as a sequence prediction problem. Given a source line, TRACER interprets it as a sequence of tokens, and attempts to predict a new sequence of tokens, that hopefully correspond to the target source line. The framework of recurrent neural networks is utilized to implement this. However, several augmentations are required for this strategy to succeed.

In particular, given a faulty program, the task of localizing the error is itself a non-trivial task. Indeed, the line numbers present in compiler error messages often themselves do not pinpoint the locations where changes need to be made. Existing works often choose to avoid this issue by, for example, expecting a neural network to jointly perform error localization and error correction [6].

However, our results show that this monolithic approach can overwhelm the underlying neural network architecture. To remedy this, TRACER performs error localization as a separate, modular step. A side advantage of delinking error localization from error correction is that TRACER is able to use different techniques for error localization and error correction which gives it more freedom whereas works such as [6] are constrained to use the same technique for the two tasks which may perform sub-optimally on one of the tasks, but nevertheless bring down the performance of the entire system.

#### 3.1 Error Localization

While the compiler error messages report the exact line of the code which resulted in an error state during compilation, this isn’t necessarily the same line where repair needs to be performed. This problem has been addressed by prior work in different ways. [7] targets the exact line number reported by compiler. [21] performs a brute force replacement, starting from the first-statement to the last; relying on their model to regenerate the correct statements, while fixing incorrect ones. [6] trains a deep-network on the entire source-program (encoded with line-numbers) to generate a ranked list of potential lines to focus their repair on.

<table>
<thead>
<tr>
<th>Lab</th>
<th># Prog</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 1</td>
<td>524</td>
<td>Hello World</td>
</tr>
<tr>
<td>Lab 2</td>
<td>1643</td>
<td>Simple Expressions</td>
</tr>
<tr>
<td>Lab 3</td>
<td>1015</td>
<td>Simple Expressions, printf, scanf</td>
</tr>
<tr>
<td>Lab 4</td>
<td>901</td>
<td>Conditionals</td>
</tr>
<tr>
<td>Lab 5</td>
<td>1104</td>
<td>Loops, Nested Loops</td>
</tr>
<tr>
<td>Lab 6</td>
<td>1098</td>
<td>Integer Arrays</td>
</tr>
<tr>
<td>Lab 7</td>
<td>1232</td>
<td>Character Arrays (Strings) and Functions</td>
</tr>
<tr>
<td>Lab 8</td>
<td>1023</td>
<td>Multi-dimensional Arrays (Matrices)</td>
</tr>
<tr>
<td>Lab 9</td>
<td>660</td>
<td>Recursion</td>
</tr>
<tr>
<td>Lab 10</td>
<td>466</td>
<td>Pointers</td>
</tr>
<tr>
<td>Lab 11</td>
<td>453</td>
<td>Algorithms (sorting, permutations, puzzles)</td>
</tr>
<tr>
<td>Lab 12</td>
<td>726</td>
<td>Structures (User-Defined data-types)</td>
</tr>
<tr>
<td>Exam</td>
<td>3427</td>
<td>Mid-term and End-term programming tests</td>
</tr>
<tr>
<td>Practice</td>
<td>9003</td>
<td>Practice problems released regularly</td>
</tr>
</tbody>
</table>
For error localization, TRACER makes a useful observation: in our dataset of introductory C programs where single-line edits were performed by students, the location of the edits lay very close to the line where the compiler flagged an error. In 87.79% of the cases, the correct source-target pairs were located at a distance of <= 1 from the line number reported in the compiler error message i.e. at, immediately above, or immediately below the compiler-reported line.

This suggests a surprisingly simple strategy for error localization: obtain a line number \( l \) from the compiler error message (recall that we are considering single-line edit programs for now) and simply consider the line numbers \( l - 1, l, l + 1 \) as candidate lines to attempt repair. This is the strategy adopted by TRACER for error localization.

What is more surprising is that this simple approach achieves almost the same error localization accuracy as much more involved techniques in literature. For instance, [6] utilize a deep network to perform error localization and report 87.5% accuracy in correctly including the location of the error in one of their top-5 predicted lines. TRACER achieves slightly higher accuracy using a technique that is much simpler.

### 3.2 Source and Target Abstraction
The second point of departure that TRACER makes from the state-of-the-art is in not processing source and target lines in the form given in the student programs. Techniques such as recurrent neural networks operate with a static vocabulary.

As a result, supplying student programs directly to these networks requires all possible identifier/literal names possibly used by students to be included in the vocabulary which not only blows up the vocabulary size, but also creates problems for extending these approaches to newer offerings of the course where students may use hitherto unused identifier names and literals.

To remedy this problem, TRACER takes source-target line pairs and processes them by replacing all literals and identifier/variables with abstract tokens representing their corresponding types. The types are inferred using LLVM (the back-end for the Clang compiler suite) [11], a standard static analysis tool.

However, there are exceptions to the above rule. The names of keywords and some standard identifiers/library-functions such as \{printf, scanf, malloc, NULL\} are retained during abstraction. User defined function names are replaced by a generic token FUNC. A special token called INVALID is used for those literals and identifiers for which static analysis is unable to reveal the type.

The character/string literals are abstracted out to remove all text. Single/double quotes, as well as any format specifiers (such as \{printf, scanf, malloc, NULL\}) and character escape-sequences (such as \{\n, \\}) are retained. For example, line number 5 in Figure 1 would be abstracted out as

\[
\text{printf("ans=%d", a+10); \rightarrow printf("%d", INT+LITERAL_.\));}
\]

These exceptions to the abstraction rule exist since several errors made by students, especially in the initial days, involve the printf and scanf functions where it is crucial to retain the format string as is, to be able to identify and fix the error. For instance, for the programs in Figure 1, the source and target lines would be abstracted as follows

\[
(1) \text{(Source) scanf("%d",a); \rightarrow scanf("%d",INT);}
\]

\[
(2) \text{(Target) scanf("%d",&a); \rightarrow scanf("%d", & INT);}
\]

These abstracted lines are called respectively, the Abstract Source Line and the Abstract Target Line. For sake of simplicity, we will often refer to them as simply the Source and the Target. Table 1 lists several actual source and target pairs from our dataset.
We note that this technique of abstracting out literals is prevalent in literature. However, previous works such as [6] replace all identifiers/variables, including string-literals, with generic distinct tokens (an un-typed ‘ID_x’). This precludes their ability to fix semantic-errors (which may be type-specific) and errors involving formatted strings (e.g. {printf,scanf} errors) which are very commonly faced by beginners. TRACER performs a much more nuanced abstraction that retains type-specific information that help it address a much wider range of errors.

3.3 Abstract Source-Target Line Translation

As mentioned before, TRACER performs repair by treating the abstract source and target lines as sequences of tokens and attempting to predict the target sequence using the source sequence. The tokens include reserved keywords such as {while, double}, standard library functions such as {printf, scanf} and abstract tokens such as {INT, FUNC, FLOAT} as discussed in the previous subsection. This sequence translation is performed using Recurrent Neural Networks (RNNs) which are described in detail in Section 4. In fact, for any source sequence, the RNN architecture is able to provide multiple suggestions for the target sequence. This is beneficial in allowing a graceful degradation of the system. In our experiments, we observed that even if the top-ranked repair presented by the RNN is not the most appropriate, usually the second or the third ranked suggestions do correspond to the most appropriate fix. TRACER uses a finely tuned encoder-decoder model with attention and long short-term memory mechanisms enabled. Details of parameter settings and tuning for our method are described in Section 5.

3.4 Target Recommendation and Concretization

TRACER offers recommended repairs in two formats – abstract and concrete (see Table 1). The abstract recommendations are obtained directly from the RNN framework by employing a beam search to obtain 5 target sequences with the highest scores. Note that these abstract sequences still contain tokens like {INT, FUNC, FLOAT}. Now, the abstract recommendations may themselves be offered to students as hints and constitute valuable feedback. It may be argued that if given the actual corrected code, the student has no incentive to explore why did s/he make an error and what was the key to fixing the error. However, if only the abstract form of the repair/solution is provided as a hint, then the student is compelled to map the abstraction onto his/her own program which, in many cases, reveals what was the error in the source program, thus fulfilling several didactic goals.

However, TRACER goes one step further. Given recommended repairs in abstract form, TRACER can generate non-abstract versions of these repairs that can be applied to the original buggy program to produce code that actually compiles. This is done by performing concretization, a process which approximately reverses the abstraction step. We observe that our concretization technique is able to convert abstract versions of the target sequence to valid compilable code in 95% of cases in our dataset.

Table 3: An example of the concretization process. The incorrect source line is \(xyz=4\) where \(xyz\) was not declared as an integer variable and a semi-colon is missing. TRACER correctly generates the abstract form of the repair \(\text{INT} = \text{LITERAL}_1:\). However, multiple concretizations, all of whom produce valid compilable code, are possible. For example, if the identifiers \(i,j\) were declared as integer variables, then \(i=4;\) and \(j=4;\) are both valid concretizations.

<table>
<thead>
<tr>
<th>source-line</th>
<th>(xyz)</th>
<th>4</th>
<th>source-abstraction</th>
<th>INVALID</th>
<th>LITERAL_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete-line</td>
<td>(i)</td>
<td>4</td>
<td>recommended-abstraction</td>
<td>INT</td>
<td>LITERAL_1</td>
</tr>
</tbody>
</table>

Each successful match with source-abstraction is replaced with corresponding concrete-code from (non-abstract) source-line. For mis-matches and inserts, TRACER replaces the recommended-abstract type with the closest concrete-code having the same type from the symbol table that is maintained by TRACER. An example of the concretization step is presented in Table 3.

3.5 Multiple Error Lines

Although the discussion so far has focused on programs where the fix is required on a single line, TRACER can be adapted to repair programs with errors present across multiple lines as well.

In our dataset, we observed that a large portion of programs having errors on multiple lines can be interpreted as simply multiple instances of single line errors, i.e. the errors in the multiple lines in these programs are not correlated and fixes may be applied to the lines individually. Hence, given a multi-line error program, TRACER

(1) **Error Localization**: TRACER fetches all the source-lines flagged by compiler for error, as well as the lines just above and just below those lines.

(2) **Code Abstraction**: TRACER obtains source-abstractions for the above source-lines.

(3) Abstract Code Repair Prediction: TRACER executes the RNN model on these source-abstractions to get the top-5 recommended-abstractions for each line.

(4) **Concretization**: TRACER finally refines each of the recommended-abstractions to generate concrete, repaired code, and checks if the compiler error associated with that particular line disappears on applying the repair. If so, the concrete-code is retained as the fix for the corresponding source-line.

We consider a multi-repair done as outlined above, to be successful, only if all compiler errors are resolved as a result.
Table 4: Performance Metrics. TRACER uses some of the most stringent performance metrics to ensure that students receive recommendations that suggest fixes that correct the actual mistake they are making, and not merely remove compilation errors.

<table>
<thead>
<tr>
<th>Performance Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prec@k, k ∈ {1 . . . 5}</td>
<td>Percentage of abstract source-target pairs where the top k abstract recommendations output by TRACER contain the abstract target line.</td>
</tr>
<tr>
<td>SPrec@k, k ∈ {1 . . . 5}</td>
<td>Percentage of abstract source-target pairs where the top k abstract recommendations output by TRACER contain any abstract target line which has the same abstract source line.</td>
</tr>
</tbody>
</table>

![Figure 4: The schematic of a recurrent neural network](image)

3.6 Performance Measures

We evaluate TRACER’s performance with performance measures that are derived from those used in information retrieval, ranking, and recommendation systems. These are some of the most unforgiving performance measures and we are not aware of their prior use in the program repair domain. The first performance measure we use is Precision at the Top (dubbed Prec@k). For every source-target pair, Prec@k gives a unit reward if the abstract target line is a part of the top k abstract recommendations returned by TRACER. Note that Prec@1 is an extremely stringent measure that accepts nothing but the exact solution expected by the student (in abstract form).

We also report performance on Smoothed Precision at the Top (SPrec@k). This performance measure takes into account the fact that upon abstraction, multiple source lines may map to the same abstract source line and consequently, that abstract source line may now map to multiple abstract target lines. Table 4 describes these performance measures.

4 RECURRENT NEURAL NETWORKS (RNN)

Recurrent neural networks (RNNs) have emerged as the learning model of choice in several areas that involve sequence modelling tasks such as natural language processing, speech recognition, image captioning etc. RNNs differ from classical neural networks in having an internal state and feedback loops into the network. The statefulness of the model and the ability to pass that state onto itself allow RNNs to effectively model sequences of data. Figure 4 shows the schematic of an RNN performing sequence translation.

Let \{x_t\}, \{y_t\} etc denote strings over a shared alphabet \(\Sigma\) i.e. for all \(t \geq 1\), we have \(x_t, y_t \in \Sigma\). In their simplest form, RNNs are trained to learn a language model by predicting the next token in a sequence correctly. An output sequence \(\{y_t\}\) is predicted as a function of the input sequence \(\{x_t\}\). More specifically, after observing \(\{x_1, x_2, \ldots, x_{t-1}\}\), the RNN predicts a token \(y_t\). The RNN is trained to ensure that \(y_t = x_t\).

In order to perform this prediction, the RNN maintains an internal state \(h_t \in H\) that also evolves over time. Usually \(H \subset \mathbb{R}^k\) is a set of real vectors of dimensionality \(k\). Let the size of the vocabulary be \(|\Sigma| = S\). Each token in \(x \in \Sigma\) is represented as a \(d\)-dimensional vector (which we also denote as \(x\) by abusing notation). This representation is usually learned as well using techniques such as Word-to-Vec etc.

The prediction is made as a function of the hidden state and the hidden state itself evolves as a function of the previous hidden state and the current input as follows:

\[
h_t = f_H(W_{hx}x_t + W_{hh}h_{t-1})
\]

\[
y_t = \text{SELECT}(f_O(W_{yh}h_t)),
\]

where \(f_O, f_H\) encode activation functions that map reals to reals (when applied to vectors, the activation functions act in a coordinate-wise manner) and \(W_{yx} \in \mathbb{R}^{S \times k}\), \(W_{hh} \in \mathbb{R}^{k \times k}\), \(W_{hx} \in \mathbb{R}^{k \times d}\) are matrices. The SELECT operation simply chooses the coordinate of a vector with the largest value. Since \(f_O(W_{yh}h_t)\) is a \(|\Sigma|\)-dimensional vector, the SELECT operation returns a token in \(\Sigma\).

Unequal Sequence Lengths: A specific hurdle that we face while using RNNs for program repair is that of input and output sequences being of unequal length. This can arise due to the fix in the incorrect program requiring insertions and/or deletions of tokens (for example, see Figures 1 and 2). The Encoder-decoder model [4, 9, 23] is a generic solution to this problem that employs two components, an encoder and a decoder. The encoder operates by using the input sequence to generate a sequence of hidden states.

\[
h_t = f_H(W_{hx}x_t + W_{hh}h_{t-1})
\]

At the end of the sequence, an intermediate representation, known as the context vector is computed as a function of the hidden state sequence \(\{h_t\}\).

\[
c = q(h_1, \ldots, h_T)
\]

The decoder generates a fresh sequence of (still hidden) states \(\{s_t\}\) using the context vector and previous hidden states. These states are used to generate an output sequence \(\{y_t\}\).

\[
p(y_t | y_1, \ldots, y_{t-1}, c) = g_O(y_{t-1}, s_t, c)
\]

until a special delimiter token (eos) is generated, indicating a termination of the output sequence.

Handling Long Sequences: A significant hurdle to training RNNs on long sequences, such as those we encounter in our program repair application, is the problem of vanishing or exploding gradients [19]. An elegant fix to this problem is the Long Short Term Memory (LSTM) model [8] and its variants which overcome the problem by replacing hidden states by a gating mechanism, which
allows gradients to flow freely in the backpropagation-through-time algorithm.

The problem emerges in a different form when dealing with long output sequences where a single context vector becomes insufficient to predict an output sequence of arbitrary length. **Attention mechanisms** [1] overcome this problem by identifying for each output token $y_t$, a specific part of input sequence which is most relevant for predicting $y_t$. This is done by employing a separate context vector $c_t$ for predicting the output token $y_t$ instead of a uniform one. These context vectors are generated by a separate neural network that is trained jointly with the RNN. Typically a weighted combination of the hidden states of the encoder are used to generate these context vectors.

We regret our inability to describe RNN frameworks in greater detail due to lack of space and refer the reader to some excellent tutorials on RNNs [10], LSTMs [17], and attention mechanisms [3].

**Usefulness of the Error Localization Step:** Even with attention mechanisms, we found RNNs with Encoder-decoder models to struggle when trained on entire programs. This is because the decoder in such situations is heavily stressed to identify the relevant subset of the input sequence for every output token, something that makes performance go down, as well as training and prediction times to go up. However, the modular approach adopted by TRACER, that first identifies a very small (often 3) set of sentences to focus on, greatly improves the ability of the RNN framework to analyze and suggest relevant local fixes.

## 5 EXPERIMENTS

**Experimental setup:** All experiments were performed on a system with an Intel® Core™ i7 CPU 930 @ 2.80GHz × 8 CPU with 8GB RAM, and an NVIDIA GeForce® GTX 760 GPU with 2GB GPU Memory.

**Dataset:** Our dataset contained a total of 16985 (source, target) line pairs, which were further divided into training (70%), validation(10%) and test(20%) sets. Validation and test examples were randomly picked from the dataset. The most frequent 150 and 140 tokens were chosen for the source and target vocabulary, respectively for training the RNN i.e. $|\Sigma| = 150$ for the input vocabulary and 140 for the output vocabulary. Tokens not in vocabulary were replaced by a special token UNK. The maximum length of source and target sequences was found to be 80 and 82, respectively.

**Training:** We trained a supervised Encoder-Decoder model with attention mechanism on an open source Torch implementation of a standard sequence-to-sequence model with (optional) attention mechanism where the encoder-decoder are LSTMs.

The models were trained by minimizing the class negative likelihood loss. We conducted an extensive grid search to set hyperparameters: number of hidden layers for both Encoder and Decoder in {1, 2, 3, 4}, size of the LSTM hidden state in {50, 100, 200, 250, 300, 350}, and word embedding size in {50, 100, 150, 200, 250, 300}. Word embeddings were learnt jointly with network parameters. We tried both unidirectional and bidirectional RNNs with reverse source side setting. Apart from this, we used a mini-batch size of 32 for training, standard stochastic gradient descent with an initial learning rate 1 and learning rate decay of 0.5 after the 9-th epoch for a total of 35 epochs. We initialized the hidden state of the Decoder at time 0 with last hidden state of the Encoder, initialized other parameters randomly in range \([-0.1, 0.1]\), clipped gradients at magnitude 5, and used a dropout probability of 0.3 between LSTM layers. The best model based on validation perplexity was found to be a bidirectional Encoder with 2 hidden layers, a hidden state size of $k = 300$, word embedding size $d = 100$. Reversing the source sequence gave poor results.

**Prediction:** We performed a beam search with beam size 50 to select the best 20 target sequences. Out of these 20 the top 5 unique predictions were used for measuring performance according to the measures described in Section 3.6.

### 5.1 Results

We note that we performed these experiments in cross validation mode with 5 random train-validation-test splits of the data as well and found results to be very similar to those we report below.

The prediction accuracies of TRACER on a test set of 4,578 programs are reported in Table-5. TRACER is able to correctly predict the exact abstraction as the student target repair as its top suggestion in 59.6% of the instances. If we include the top 3 recommendations of TRACER, then this figure jumps to 66.6% for Prec@3 and further to 73.85% for SRec@3.

Figure-3b shows that TRACER consistently achieves high levels of Prec@1 accuracy in predicting the exact student repair, irrespective of the type of compiler error. It fails on just two error types, namely $E7$: too few args to function call and $E13$: too many args to function call. This is because presently, TRACER cannot predict the correct number of parameters for a function call in the target.

We have developed extensions of TRACER that seek to correct this limitation which will appear in future reports.

Table-1 lists a few interesting examples from our dataset on which TRACER was able to provide correct and relevant fixes. For the first two examples, TRACER makes an accurate prediction at the very top, correctly suggesting the insertion of an arithmetic operator in example #1 and switching the L and R values in example #2. For the rest of the examples, its top recommendation does not match the student fix. However, notice that TRACER’s top fix is still an appropriate fix.

In examples #4 and #5, TRACER predicts an abstract fix which is semantically equivalent to the desired fix but we nevertheless penalize it due to our stringent evaluation criteria of an exact match with the student repair. TRACER arguably produces a simpler fix than what the student devised for examples #4 and #5. TRACER is.
able to improve upon any individual student’s fixes since it learns from frequent patterns from other student submissions as well.

The most interesting case is example #3 where the student’s submission, though syntactically correct, results in a compiler warning “incompatible integer to pointer conversion passing ‘int’ to parameter of type ‘const char’.” TRACER’s suggestion is better since it does not generate any compiler warning, and is doing what is intended. However, our evaluation metrics still consider it a failure due to the mismatch with the student repair.

In future, we plan to develop better scoring mechanisms to evaluate our predictions. Nonetheless, even with such stringent metrics, TRACER is able to achieve high accuracy of 59.6% for prediction@1. For example #6, TRACER produces the source verbatim since it is unable to figure out the desired number of opening braces from the local context. In future, we plan to incorporate global context as well to predict such fixes.

5.2 End-to-End Repair

In Figure 5, we demonstrate the overall end-to-end accuracy of TRACER on our dataset of single-line errors, across all the labs. The Abstraction_Match legend denotes how many times our recommended-abstraction matched the exact target-abstraction (the abstraction of fix performed by same student), Concrete_Compile denotes what % of the erroneous programs was TRACER able to repair and compile.

Table 6 reports the compilation success rates of the multiple-error approach outlined in Section 3.5 on our single-line (errors requiring fix at a single line), multiple-line (errors requiring fix at multiple different lines), and on the dataset obtained from DeepFix [6].

TRACER is able to fix 79.27% of the programs on our testing set of single-line edit program pairs (referred to as Single_Test), while being highly relevant at the same time (as demonstrated in Table 5). TRACER is also able to successfully repair about 44% of programs with multiple-line errors obtained from our course (denoted as “Multiple” in Table 6) and on the collection of programs used in DeepFix [6]. Note that DeepFix, the current state-of-the-art was able to achieve only 27% compilation success rate on the same set of programs.

Invoking the deep-network takes the maximum amount of time in this entire setup, while abstraction/concretization and compilation steps take an order of milli-seconds time on average. Hence to ameliorate this, we cache the frequently looked-up abstraction translations during training phase. This reduces the time taken for the entire repair process to just 1.66 seconds on average per erroneous program, during the testing phase.

Table 6: Overall accuracy of TRACER on three different datasets. Single_Test is the test set on single-line edit program pairs. Multiple is the set of programs requiring repairs on multiple lines. DeepFix is the dataset used by [6]. On all datasets TRACER gives compilable code on a large fraction of programs. On the DeepFix dataset, the method of [6] itself offers only 27% compilation success whereas TRACER offers a much higher 44% success rate.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Programs</th>
<th>#Compiler-Errors</th>
<th>TRACER Repair %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single_Test</td>
<td>4,578</td>
<td>4,853</td>
<td>79.27</td>
</tr>
<tr>
<td>Multiple</td>
<td>17,451</td>
<td>24,255</td>
<td>43.67</td>
</tr>
<tr>
<td>DeepFix [6]</td>
<td>3,226</td>
<td>13,266</td>
<td>44.54</td>
</tr>
</tbody>
</table>

Figure 5: Overall accuracy of TRACER, shown per lab. TRACER is robust to handle compiler errors occurring across all the labs. Even on the erroneous programs from labs dealing with advanced concepts such as pointers and recursion, TRACER is able to consistently achieve high repair rate.

non-trivial fraction of buggy programs, but this may not be an acceptable fix.

The approach used by DeepFix [6] to repair common C programming errors comes closest to our work. DeepFix learns a seq2seq Neural Network on the entire program, and features an error localization module which, with the help of an Oracle, can attempt to resolve multi-line errors by making multiple passes. TRACER on the other hand focuses on single-line errors and achieves a high compilation accuracy, while being highly relevant to the student target. In fact on the very dataset used by [6], TRACER offers significantly higher compilation success rates than DeepFix itself.

HelpMeOut [7] gives relevant examples of solutions for a compilation error by searching a database for the same error encountered by other students. It provides both the erroneous line of the other student, as reported by compiler, as well as the modified line which resulted in successful compilation. However, unlike TRACER, the repair is picked from a database and is not tuned to the erroneous source program. Moreover, the approach provides the student the fix directly, which can be suboptimal in terms of learning outcomes. TRACER instead learns the repair from the submissions of the other

6 RELATED WORK

To the best of our knowledge, TRACER is the first approach to perform targeted repair where we learn and predict the exact fix for a given bug, instead of predicting it as a side-effect of not matching with a generic correct grammar. Prior works [2, 6, 20] learn repairs by observing generic correct programs, but do not have a well defined criterion to evaluate the quality of the repairs. For example, simply deleting the erroneous line makes the error go away in a
students, and suggests the exact abstract/concrete fix tuned for the erroneous source program.

A recent study [27] has concluded that program repairs can also be used by human graders as the graders are experienced programmers. In that study, repairs were semantic repairs and were generated by four different tools, namely GenProg [12], AE [26], Angelix [14], and Prophet [13]. The positive impact of compilation error repairs on (automated) grading is also reported by Gradelt [18] that uses simple rewrite rules to repair compilation errors. The repairs generated by TRACER can further improve the performance of systems such as Gradelt and a matter of an ongoing investigation.

7 FUTURE WORK

Our experiments confirm that TRACER performs very well on errors where repair can typically be obtained by looking at a local context, for example, inserting a token or restructuring an expression. However, TRACER does not yet take into account the global context, e.g., the number of parameters of a function defined away from its call, opening/closing a missing brace, undeclared variables etc. Some of these global context errors occur in the most frequent list and can be handled by a separate technique which runs in tandem with TRACER. Future versions of TRACER are planned to have these features to widen its scope.

Also, our evaluation metrics currently do not discount minor syntactic variations, e.g. \( a \leftarrow b \) vs \( \{a \leftarrow b) \), while computing the mismatch between tool repair and the student repair. We believe a more moderate performance measure will give us a more realistic picture of the performance of TRACER and other such error-correction tools. Finally, it is worthwhile to combine TRACER with semantic repair tools such as Prophet [13] to further refine the repairs. TRACER is currently being deployed on a live offering of the C programming course and a study investigating its effects on student experience will be released soon.

8 CONCLUSION

We presented TRACER, a tool to generate targeted repairs aimed at novice programmers. TRACER offers extremely accurate recommendations for fixes and on more than 4000 test programs with single-line errors, its top three recommendations include the correct repair 74% of the time. Even on more than 17000 programs with errors on multiple lines, TRACER offers a compilation success rate of more than 40%. TRACER takes only several milliseconds to make its recommendations, making it suitable for live deployment. Although we focused on the C programming language in this paper, TRACER can be easily ported to other programming languages such as C++, Java, etc. Given enough data, TRACER can also be used predict fixes for compilation errors in the programs in future offerings of a course. TRACER’s performance on stringent correctness criterion strengthens our claim that repairs suggested by it are close to the fixes performed by the students themselves. As a result, these repairs are of more didactic value than the compiler produced error messages or repairs that aim to only remove compile-time errors. High accuracy and student-friendliness of the repairs generated by TRACER make it an attractive virtual teaching assistant for programming courses offered at a massive scale.

REFERENCES

[21] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Com-