Program Transformation via Natural Language Instructions

Celine Lee

Abstract

The purpose of this project is to design a code modification system. The proposed system takes in a pair of existing code snippet and natural language utterance requesting some modification to the code. The system output would be the code snippet, modified according to the natural language request. In this project, we explore two model approaches to the task: (1) a Codex-based system, and (2) designing a custom transformation DSL.

1. Introduction

The process of developing software frequently involves iteratively modifying code, whether rewriting code segments to change semantics or refactoring code bases to change syntactical implementation details. It is also generally-accepted good practice for programmers to annotate their source code with natural language comments. Additionally, when sharing code in repositories such as GitHub, programmers add a natural language description to each code push to label the modifications made to the code. Natural language again closely accompanies code in code forums such as StackOverflow and Github issues as programmers use natural language to communicate the goals of and issues in their code. This repeated association between natural language and source code semantics sets the stage for this project and other similar works, where we can examine how natural language may be used as a tool to perform software development.

In this project, we use natural language requests as a specification to transform program source code. Example usage of such a system may be to reduce manual human labor in code refactoring or to accelerate program development. From such automated transformations, programmers can continue to iterate atop the baseline implementation either by continuing to use the proposed program transformation system or by manually adjusting small details.

2. Related Work

2.1 Alignment of Source Code and Natural Language

A number of works have examined alignments of code and natural language. Examples include how to mine aligned code and natural language pairs from StackOverflow [1], generally associating natural language comments and source code entities [2], and generating [3, 4, 5] or updating [6] natural language descriptions from source code or source code changes.

2.2 Semantic Parsing

Semantic parsing is the natural language (NL) processing task generally concerned with converting NL *utterances* (i.e., the smallest unit of speech) to structured logical forms. Once done, these logical forms can then be utilized to perform various machine tasks such as question answering [7, 8, 9, 10, 11], machine translation [12, 13], or code generation [14, 15, 16, 17], amongst other things.

Given an NL input, a NL semantic parsing system generates its semantic representation as a structured logical form (e.g. source code). These logical forms are also known as meaning representations or programs. To generate these outputs, semantic parsing pipelines are usually trained by some learning algorithm to produce a distribution over the output search space. They then search for a best scoring logical form in that distribution. This logical form can then be executed against some environment to carry out an intended task (e.g. run the source code on a machine).

For semantic parsing for code generation, the end logical form of the system will usually be some form of software code. This may be the direct output program [16, 18, 7, 19] or an intermediate representation of code [14, 20, 21, 22, 8], which can subsequently be fed into a program synthesis component to generate syntactically sound and functionally correct code.

Semantic parsing has evolved from early hand crafted rules in the 1970s [23, 24, 9, 25, 26] to the rise of learned grammars via statistical learning methods in the late 1990s and early 2000s [19, 7, 27, 28, 29, 30, 31, 32], to modern neural seq2seq semantic parsing efforts [8, 12, 14, 22, 16, 33, 34].

Semantic parsing can also be deployed in a dialog framework [35], where the user and system can communicate back-and-forth to semantically parse then represent the user intention as a dataflow graph that can be modified throughout the course of the conversation until the user goal is achieved.

2.3 Autoregressive Language Models

Deep learning techniques such as autoregressive language models that predict the next token in a sequence have seen a rise in popularity recently with the increased access to massive amounts of data and computational power. Especially, recent efforts have gone into designing pre-trained models for processing programming language. Pre-trained models help address the redundancy of training different models for similar problems by training a model to solve some machine learning problem, then fine-tuning that pre-trained model on the different task. The fine-tuning has fewer rounds of training, allowing us to build accurate models for different tasks while saving training computation cost and time.

CodeBERT is a pre-trained model inspired from Transformers-based neural architecture that learns general-purpose representations for NL and PL [36]. Program and Language BART (PLBART) [37] is another a sequence-to-sequence model pre-trained on a large collection of unlabeled Java and Python functions and associated NL text to learn multilingual representations of syntax and semantics for PL and NL applications. PLBART uses an additional normalization layer on top of both the encoder and decoder architecture used in the BART-BASE pre-trained model [38]. Other models feed sequentialized representations of program ASTs into the model in order to inject syntactic structure of code into the model when predicting the next token of program source code [39]. GraphCodeBERT also factors in the inherent semantic structure of code by using code data flow information rather than the AST for the input [40]. This structure-aware approach uses graph-guided masked attention functions on the Transfomer-based architecture to improve measured performance on various code tasks, as compared to CodeBERT and other pre-existing pre-trained models.

The Generative Pre-trained Transformer (GPT) family of models [41, 42, 43] implements a transformers model with attention, the most recent of which, GPT-3, has 175 billion parameters. The GPT-n models can produce human-like text and be fine-tuned with few examples to perform most natural language processing task with fairly high accuracy. Codex [44] is a GPT language model fine-tuned on source code from GitHub repositories to generate programs from natural language docstrings.

2.4 Automated Program Repair

Automated program repair (APR) [45] is the task of automatically repairing software, thereby ideally reducing the work of human engineers while maintaining program usability and avoiding software regression. Works in this space have used prior bug fixes and surrounding code context to modify a buggy program's AST [46] and neuro-symbolic transformation networks to predict bug locations and fixes without access to the code's intended correct behavior [47].

Some APR works have leveraged natural language in buggy program error messages to model reasoning about automated repair [48]. TFix [49] is another end-to-end text-to-text system that fixes buggy code without labels by pre-training a model on natural language then fine-tuning it on generating code fixes.

3. Model 1: Using Codex

In the first system designed for this project, we leverage the modeling power of Codex, a language model created by OpenAI using artificial intelligence to ”translate natural language to code” [50].

Codex [44] is an autoregressive language model pre-trained on massive amounts of text and fine-tuned on source code from GitHub. As an autoregressive model, its objective is to output the next most likely token in some generated sequence. The model is prompted by some input string of tokens and the model outputs a most likely following sequence of tokens. In source code, code typically follows natural language docstrings describing the objective of the code, so this model leverages the learned text representations of GPT and learned patterns from fine tuning on GitHub repositories to generate code from descriptive docstrings.

We want to see how we can use this natural language to code generation to perform natural language to code transformation. All code and data for this section is available at `https://github.com/celine-lee/transform_code`. The repository is private right now, but I am happy to share access with class instructors; just let me know.

3.1 Data

The data used for the Codex-based code transformation system comes from Panthaplackel et al.'s related but reverse paper 'Learning to Update Natural Language Comments Based on Code Changes' [51]. The dataset is collected from commit history of certain open-source Java software projects. Each example corresponds to a concurrent method and comment update.

From their data, we extract data points of old comment, old code, new comment, and new code. We then formulate Codex input prompts from the data and use the Codex output as described in the following section.

3.2 System

The Codex-based system is designed to work with the autoregressive nature of the Codex model and the old-comment-new-comment structure of the data. We do not further fine-tune Codex for our task, so the Codex-based system is broken down into four stages: (1) pre-process the data, (2) generate the CURL request to the Codex API, (3) process the returned Codex output, (4) evaluate the result.

3.2.1 Pre-process the data

We process the data to extract old code, new code, old comment, new comment, and a span-style code diff. We maintain a JSON dict mapping unique commit ID to this five-tuple.

Then from the old code and new code, we extract the longest common prefix string. This will be used at the end of the input prompt, as described in generating the CURL request.

3.2.2 Generate the CURL request

In addition to the input prompt itself, the Codex API has a number of parameters that can be used to configure the query.

The input prompt is formed as follows:

```
// <old_comment>
<old_code>

// <new_comment>
<shared_prefix(old_code, new_code)>
```

By providing the old comment and old code, we give Codex a prior example of what we want the output code to look like. Then, by providing the new comment, we prompt for more (modified) code. Additionally providing the shared prefix, i.e. part of the "answer," is intended to further guide Codex to produce slightly transformed code rather than entirely different code.

Other parameters that we set or provide configuration options for are temperature, log probabilities, maximum number of tokens, and stop strings. Temperature is the sampling temperature, where higher values encourage the model to take more risks and come up with lower-but-still-high-probability tokens and lower values (0) will encourage safer prediction by taking the argmax token each time. We found slightly improved performance with temperature set to 0.5. Log probabilities is an integer parameter determining how many of the highest log probability tokens to return from the model. I set log probabilities to 4. Maximum number of tokens places an upper limit on how many tokens Codex will generate off of a single prompt. This max number of tokens, assuming that we do not concern ourselves with stop words, will generally be the maximum difference between number of tokens from the answer solution and the number of tokens in the shared prefix. Stop strings are character / token sequences where we want the model to stop generating more tokens. We set the stop sequences to the start of a Java comment or a Python comment: `//`, `/*`, `#`. The Python comment indicator, `#`, is included for completeness because Codex is specially geared for generating Python code. Additionally, Codex will typically prepend a commented text description before writing additional code past the scope of the current method being written, so this inclusion of comment indicators as stop tokens serves to discourage Codex from babbling more examples.

3.2.3 Process Codex output

The output string from Codex is then appended to the shared prefix string in order to generate the complete transformed code snippet.

3.2.4 Evaluate the result

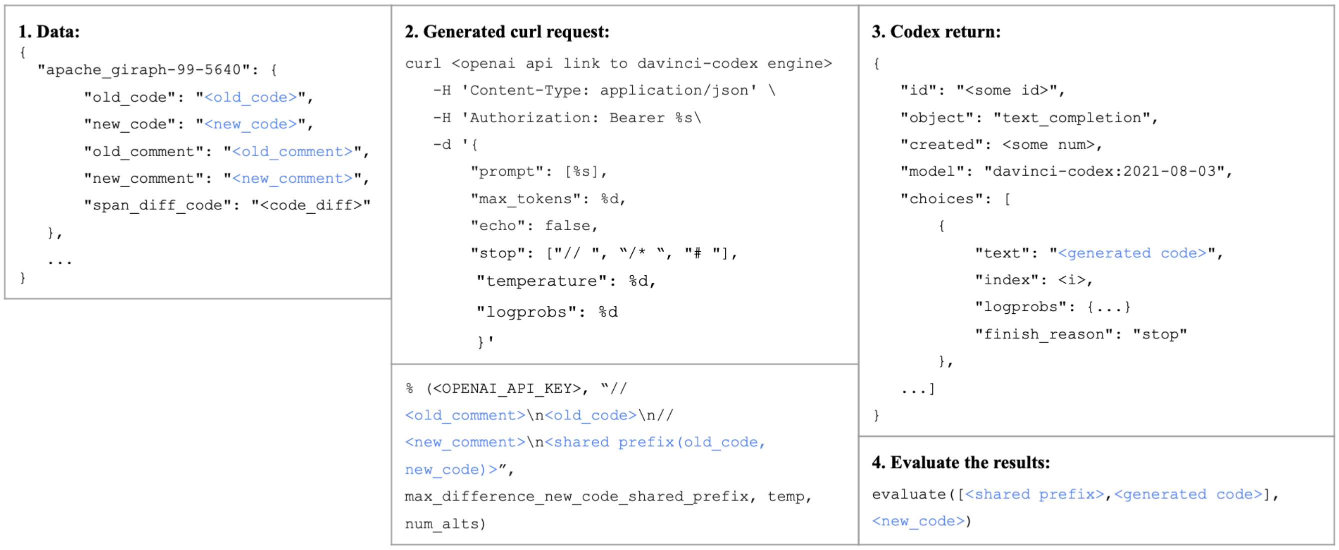
The Codex system is evaluated on a test suite of code transformation cases. The test suite is manually selected to refactoring examples such as modifying the input or output object types and alternative library method selection, as well as other semantics-altering examples ranging from modifying just a couple of tokens to modifying multiple lines of code in the method. Evaluation metrics include exact match, exact match excluding variable and method names, and manual inspection for semantic equivalence.

3.3 Results and Analysis

Results on a test suite of 17 examples are shown in Table 1 for temperature values of 1.0, 0.5 and 0.3, and top-4 log probability tokens. The number shown in the table is the number of test cases (out of 17) that pass the given test. Manual inspection for semantic equivalence seems to play an important role in evaluating performance of the Codex system. Figure 2 shows when Codex produces code that is syntactically distinct from the "correct" answer but semantically equivalent.

Table. 1. Codex-based system results

Temperature	Exact	Variable Flex	Manual
1.0	0	1	2
0.5	3	5	9
0.3	3	5	8



1. Data:

```
{
  "apache_giraph-99-5640": {
       "old_code": "<old_code>",
       "new_code": "<new_code>",
       "old_comment": "<old_comment>",
       "new_comment": "<new_comment>",
       "span_diff_code": "<code_diff>"
   },
   ...
}
```

2. Generated curl request:

```
curl <openai api link to davinci-codex engine>
   -H 'Content-Type: application/json' \
   -H 'Authorization: Bearer %s\
   -d '{
       "prompt": [%s],
       "max_tokens": %d,
       "echo": false,
       "stop": ["// ", "/* ", "# "],
        "temperature": %d,
        "logprobs": %d
        }'
```

```
% (<OPENAI_API_KEY>, "//
<old_comment>\n<old_code>\n//
<new_comment>\n<shared prefix(old_code,
new_code)>",
max_difference_new_code_shared_prefix, temp,
num_alts)
```

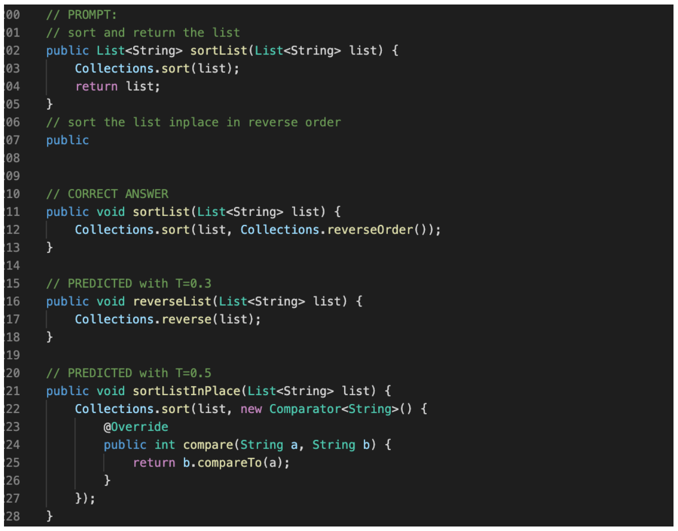
3. Codex return:

```
{
   "id": "<some id>",
   "object": "text_completion",
   "created": <some num>,
   "model": "davinci-codex:2021-08-03",
   "choices": [
       {
           "text": "<generated code>",
           "index": <i>,
           "logprobs": {...}
           "finish_reason": "stop"
       },
   ...]
}
```

4. Evaluate the results:

```
evaluate([<shared prefix>,<generated code>],
<new_code>)
```

Fig. 1. States of Codex-based system.



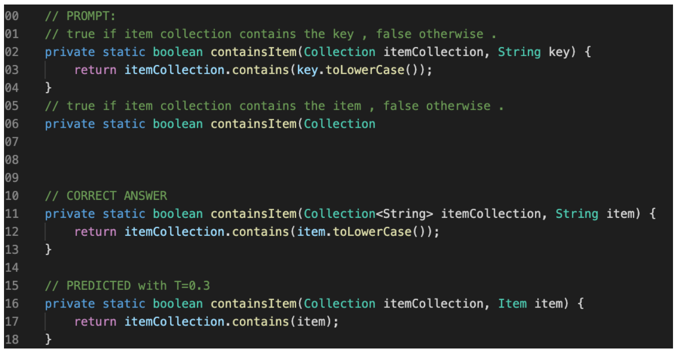
```java
// PROMPT:
// sort and return the list
public List<String> sortList(List<String> list) {
    Collections.sort(list);
    return list;
}
// sort the list inplace in reverse order
public


// CORRECT ANSWER
public void sortList(List<String> list) {
    Collections.sort(list, Collections.reverseOrder());
}

// PREDICTED with T=0.3
public void reverseList(List<String> list) {
    Collections.reverse(list);
}

// PREDICTED with T=0.5
public void sortListInPlace(List<String> list) {
    Collections.sort(list, new Comparator<String>() {
        @Override
        public int compare(String a, String b) {
            return b.compareTo(a);
        }
    });
}
```

Fig. 2. Codex produced semantically equivalent code.

In order to understand the performance of this system, we summarize common causes of failure and analyze some specific test cases. Common causes of failure in these test cases are: (1) misunderstanding user intent in the natural language request, (2) using or building off of the old code instead of rewriting it, and (3) not knowing what functions and variables it had available from the surrounding code context. We examine a couple of specific cases from the test suite here.

In Figure 3, the natural language requests for the function to check whether the `Collection` contains the "item" (rather than the "key" as in the old code), and so Codex produces code that checks whether the `Collection` contains the passed-in `Item` object is in the `Collection`. However, the user intended for the method to check whether some `String` item is in a `Collection` of `Strings`.

In Figures 4 and 5, Codex built off of the old code instead of rewriting code. In Figure 4, instead of writing `GetAllInterfaces` as it should have done, Codex wrote the `getInterfaces` function that was referred to in the prompt. Additionally interesting, on line 29, the code that Codex wrote may show some indication of understanding method overloading; Codex wrote a static void method called `getInterfaces`, and in it, it referred to a class method also called `getInterfaces` but with different arguments. In Figure 5, the generated code calls the old code. It does do it correctly, as the generated code is semantically equivalent to what the user wanted; this is expected with Codex-generated code, as it is designed to write code, not necessarily overwrite code. The ability of



```
// PROMPT:
// true if item collection contains the key , false otherwise .
private static boolean containsItem(Collection itemCollection, String key) {
    return itemCollection.contains(key.toLowerCase());
}
// true if item collection contains the item , false otherwise .
private static boolean containsItem(Collection



// CORRECT ANSWER
private static boolean containsItem(Collection<String> itemCollection, String item) {
    return itemCollection.contains(item.toLowerCase());
}

// PREDICTED with T=0.3
private static boolean containsItem(Collection itemCollection, Item item) {
    return itemCollection.contains(item);
}
```

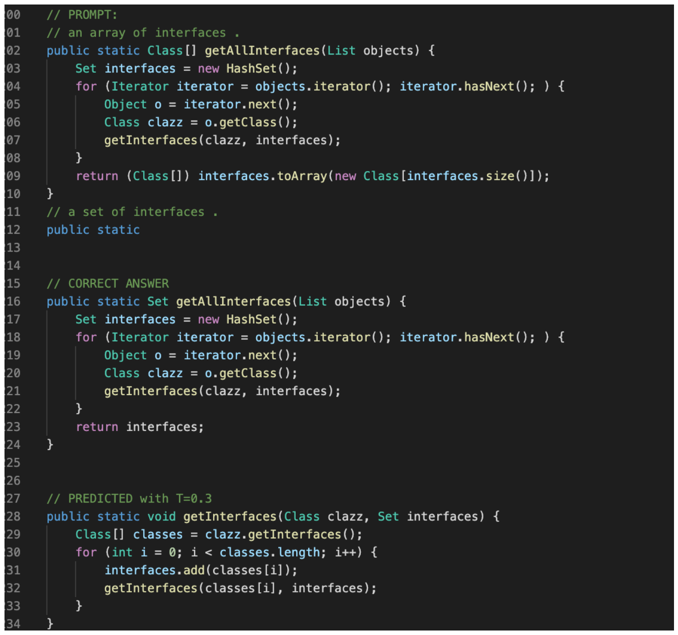
Fig. 3. Codex misunderstood user intent.

Codex to produce this result is additionally interesting because it correctly intuited how to use the old code for this slightly modified purpose.

Failure category (3) is especially interesting to examine. Codex attempts to assemble the code that it thinks is most likely to follow the prompt, but it does not know what code it has available from elsewhere in the repository that the code snippet comes from. As a result, it may incorrectly extrapolate available functions. Figure 6 illustrates one example of this. The transformed code evidently uses an updated `this.facade.getContent()` function that now returns a `ByteBuf` object instead of the original `Buffer` object. Codex has no reason to know this, so it instead calls a different `this.facade.getContentAsB()` function. The textual semantics that play into this, however, are quite interesting. We know that the transformed code should return a `ByteBuf` type object. Codex fabricated some function name ending with `asB`, which is something that human programmers commonly stylistically do: embed the return type into variants of a function using ”as”, and shorten types to the first letter of that type (e.g. `int` variables as `i`, `string` variables as `s`). This suggests that

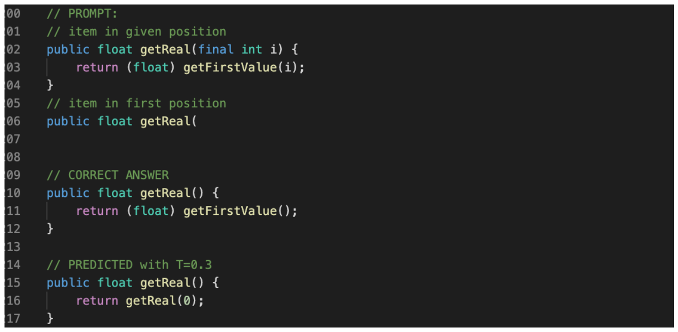
3.4 Future work

Codex has demonstrated an impressive ability to transform code as requested by a natural language utterance. We see a couple avenues for future work. One immediate avenue of exploration is an evaluation system that uses the top-logprobabilies output by Codex to see whether the Codex-based system does at least have the correct result in its top options. Also, a system that includes some attention-based token copying may help the performance of this Codex-based system manage such concepts as library functions, custom object types, and variable / method names. Another aspect to consider is that some of the test cases were not particularly well-suited for the purpose of transformation. It may be to our interest to generate a dataset de-



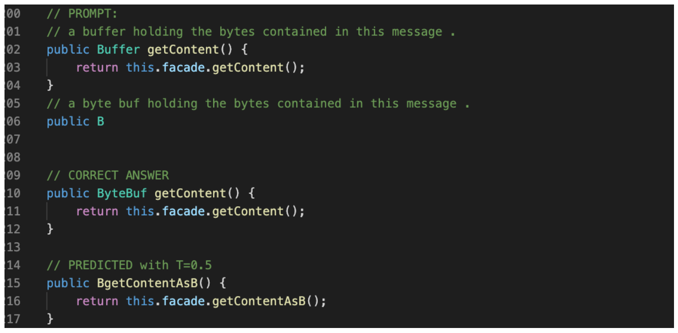
```
// PROMPT:
// an array of interfaces .
public static Class[] getAllInterfaces(List objects) {
    Set interfaces = new HashSet();
    for (Iterator iterator = objects.iterator(); iterator.hasNext(); ) {
        Object o = iterator.next();
        Class clazz = o.getClass();
        getInterfaces(clazz, interfaces);
    }
    return (Class[]) interfaces.toArray(new Class[interfaces.size()]);
}
// a set of interfaces .
public static


// CORRECT ANSWER
public static Set getAllInterfaces(List objects) {
    Set interfaces = new HashSet();
    for (Iterator iterator = objects.iterator(); iterator.hasNext(); ) {
        Object o = iterator.next();
        Class clazz = o.getClass();
        getInterfaces(clazz, interfaces);
    }
    return interfaces;
}


// PREDICTED with T=0.3
public static void getInterfaces(Class clazz, Set interfaces) {
    Class[] classes = clazz.getInterfaces();
    for (int i = 0; i < classes.length; i++) {
        interfaces.add(classes[i]);
        getInterfaces(classes[i], interfaces);
    }
}
```

Fig. 4. Codex built off old code.



```
// PROMPT:
// item in given position
public float getReal(final int i) {
    return (float) getFirstValue(i);
}
// item in first position
public float getReal(


// CORRECT ANSWER
public float getReal() {
    return (float) getFirstValue();
}

// PREDICTED with T=0.3
public float getReal() {
    return getReal(0);
}
```

Fig. 5. Codex used old code.



```
// PROMPT:
// a buffer holding the bytes contained in this message .
public Buffer getContent() {
    return this.facade.getContent();
}
// a byte buf holding the bytes contained in this message .
public B


// CORRECT ANSWER
public ByteBuf getContent() {
    return this.facade.getContent();
}

// PREDICTED with T=0.5
public BgetContentAsB() {
    return this.facade.getContentAsB();
}
```

Fig. 6. Codex is not aware of functions it has available.

signed for code transformation with better-written natural language requests rather than using updated comments in GitHub commits.

4. Model 2: Using a custom DSL

The second system designed for this project will involve writing and using a custom domain specific language (DSL) for the purpose of modifying code in some specified domain (e.g. arithmetic operations, string modification operations). The DSL, once generated from the natural language utterance, will be applied to the input source code in order to deterministically generate the transformed output source code.

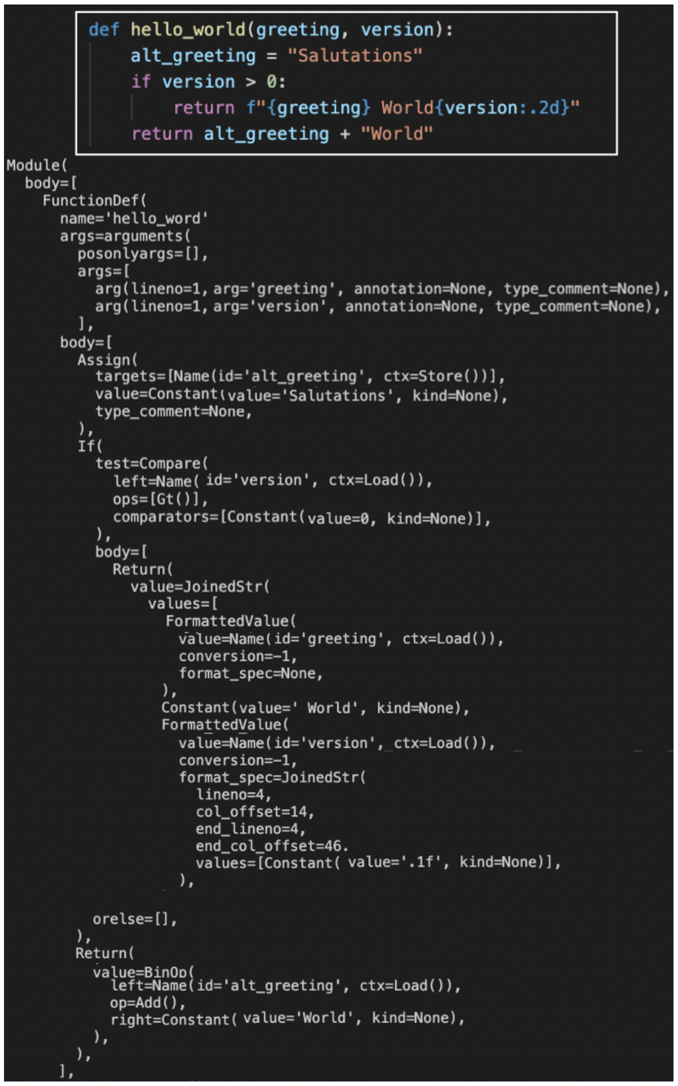
All code and data for this section is available at `https://github.com/celine-lee/transform_code/tree/main/txm_dsl`. The repository is private right now, but I am happy to share access with class instructors; just let me know.

4.1 Motivation

The motivation behind designing and using a custom DSL is twofold: (1) we need a language for modifying programs and (2) to restrict the scope of the generated programs, making the semantic parsing task from natural language utterance to machine representation an ”easier” task. With the reduced scope of potential output semantic parsing meaning representations, since they will now be in terms of the DSL rather than a whole general-purpose language such as Java as done in Model 1, we may be able to successfully explore alternative cheaper program synthesis techniques.

4.2 The DSL

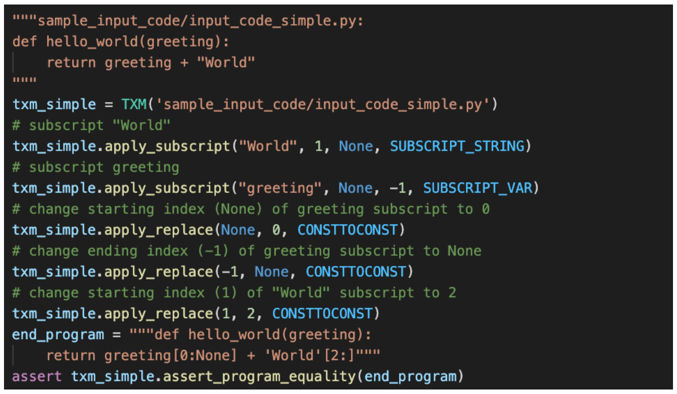
For this project, the DSL has been designed to operate over Python code. We use the Python ast library to parse Python code into its abstract syntax tree (AST). An example of a Python method and its corresponding parsed AST is shown in Figure 7. Our DSL then operates over the nodes of the parsed AST to make modifications to the code. The DSL is incomplete for the time being, but is designed to include transformations that would be useful for string and list manipulation in a Python function. As a proof of concept, in this part of the project we show that we can perform such transformations. The intention for future usage of this DSL is to use it as the grammar for a semantic parsing output meaning representation. Then we can apply machine translation or other semantic parsing techniques to translate natural language into the corresponding program in the transformation DSL.



```
def hello_world(greeting, version):
    alt_greeting = "Salutations"
    if version > 0:
        return f"{greeting} World{version:.2d}"
    return alt_greeting + "World"
```

```
Module(
  body=[
    FunctionDef(
      name='hello_word'
      args=arguments(
        posonlyargs=[],
        args=[
          arg(lineno=1, arg='greeting', annotation=None, type_comment=None),
          arg(lineno=1, arg='version', annotation=None, type_comment=None),
        ],
      body=[
        Assign(
          targets=[Name(id='alt_greeting', ctx=Store())],
          value=Constant(value='Salutations', kind=None),
          type_comment=None,
        ),
        If(
          test=Compare(
            left=Name( id='version', ctx=Load()),
            ops=[Gt()],
            comparators=[Constant(value=0, kind=None)],
          ),
          body=[
            Return(
              value=JoinedStr(
                values=[
                  FormattedValue(
                   value=Name(id='greeting', ctx=Load()),
                   conversion=-1,
                   format_spec=None,
                  ),
                 Constant(value=' World', kind=None),
                 FormattedValue(
                   value=Name(id='version', ctx=Load()),
                   conversion=-1,
                   format_spec=JoinedStr(
                     lineno=4,
                     col_offset=14,
                     end_lineno=4,
                     end_col_offset=46.
                     values=[Constant( value='.1f', kind=None)],
                   ),

          orelse=[],
        ),
        Return(
          value=BinOp(
            left=Name(id='alt_greeting', ctx=Load()),
            op=Add(),
            right=Constant( value='World', kind=None),
          ),
        ),
      ],
```

Fig. 7. Sample AST processing of Python code.



```python
"""sample_input_code/input_code_simple.py:
def hello_world(greeting):
    return greeting + "World"
"""
txm_simple = TXM('sample_input_code/input_code_simple.py')
# subscript "World"
txm_simple.apply_subscript("World", 1, None, SUBSCRIPT_STRING)
# subscript greeting
txm_simple.apply_subscript("greeting", None, -1, SUBSCRIPT_VAR)
# change starting index (None) of greeting subscript to 0
txm_simple.apply_replace(None, 0, CONSTTOCONST)
# change ending index (-1) of greeting subscript to None
txm_simple.apply_replace(-1, None, CONSTTOCONST)
# change starting index (1) of "World" subscript to 2
txm_simple.apply_replace(1, 2, CONSTTOCONST)
end_program = """def hello_world(greeting):
    return greeting[0:None] + 'World'[2:]"""
assert txm_simple.assert_program_equality(end_program)
```

Fig. 8. Add and update subscripting of strings, variables.

As of the time of writing the report, the implemented DSL is as follows:

```
string
int
node:= CONSTNODE({string, int} value)
     | VARNODE(string varname)
     | INSERT(node tree, node new, int index)
     | REMOVE(node tree, ast.Class, *int indices)
     | REPLACE(node tree, {string,int} old,
         {string,int} new, int mode)
     | SUBSCRIPT(node:tree, string to_subscript,
         int? new_low, int? new_high, int mode)
     | UPDATE_FORMATTEDVALUE(node tree,
         {string,int} fvalue, int? conversion,
         string? format_spec)
     | UPDATE_FSTRING(node tree,{*node,*int} mods)
     | more to come...
```

4.2.1 Examples

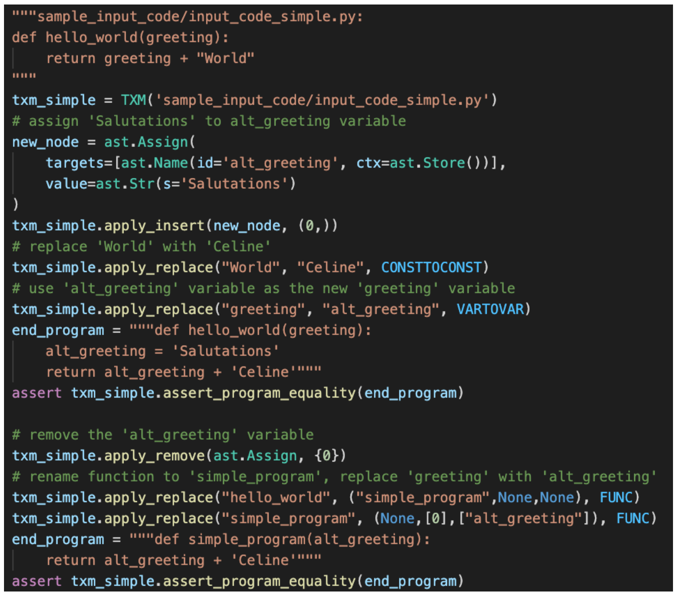
Sample usages of the DSL are shown in Figures 8, 9, 10, 11, and 11.

4.3 System design

Once the DSL has been fully realized, we can write a semantic parsing system to translate natural language requests to programs in the DSL. The DSL can then, as shown in the previous section, be used to transform Python programs. The semantic parsing system will likely be trained using neural machine translation techniques.

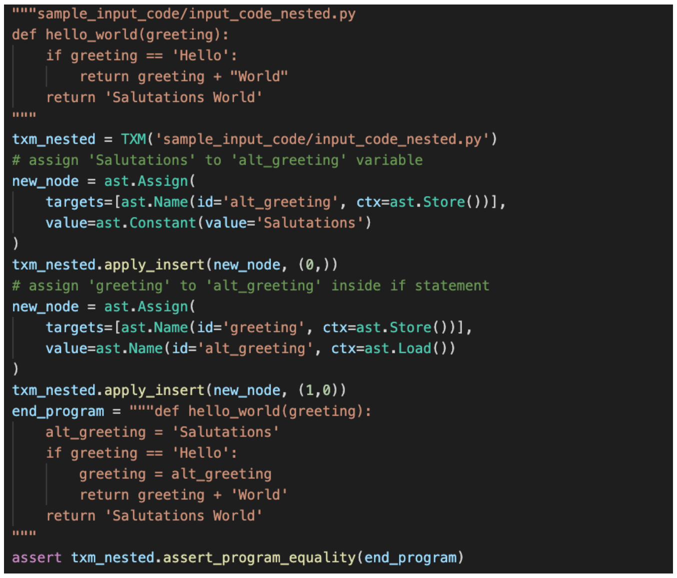
Bibliography

[1] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, “Learning to mine aligned code and natural language pairs from stack overflow,” *2018 IEEE/ACM*



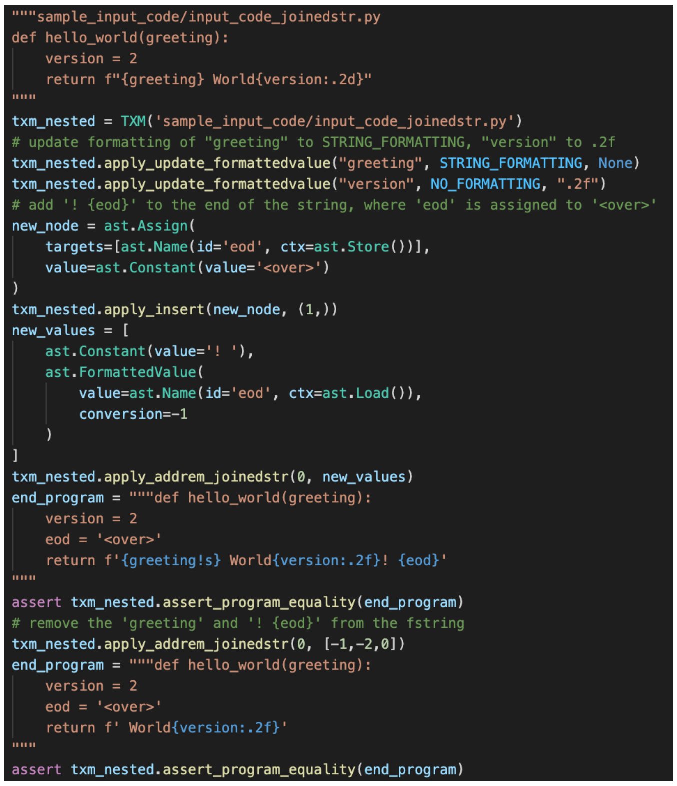
```python
"""sample_input_code/input_code_simple.py:
def hello_world(greeting):
    return greeting + "World"
"""
txm_simple = TXM('sample_input_code/input_code_simple.py')
# assign 'Salutations' to alt_greeting variable
new_node = ast.Assign(
    targets=[ast.Name(id='alt_greeting', ctx=ast.Store())],
    value=ast.Str(s='Salutations')
)
txm_simple.apply_insert(new_node, (0,))
# replace 'World' with 'Celine'
txm_simple.apply_replace("World", "Celine", CONSTTOCONST)
# use 'alt_greeting' variable as the new 'greeting' variable
txm_simple.apply_replace("greeting", "alt_greeting", VARTOVAR)
end_program = """def hello_world(greeting):
    alt_greeting = 'Salutations'
    return alt_greeting + 'Celine'"""
assert txm_simple.assert_program_equality(end_program)

# remove the 'alt_greeting' variable
txm_simple.apply_remove(ast.Assign, {0})
# rename function to 'simple_program', replace 'greeting' with 'alt_greeting'
txm_simple.apply_replace("hello_world", ("simple_program",None,None), FUNC)
txm_simple.apply_replace("simple_program", (None,[0],["alt_greeting"]), FUNC)
end_program = """def simple_program(alt_greeting):
    return alt_greeting + 'Celine'"""
assert txm_simple.assert_program_equality(end_program)
```

Fig. 9. Insert, replace, and remove variables and strings.



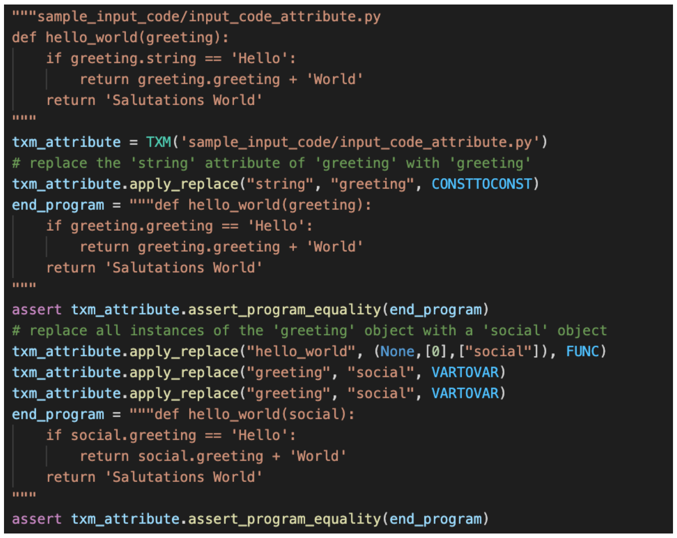
```python
"""sample_input_code/input_code_nested.py
def hello_world(greeting):
    if greeting == 'Hello':
        return greeting + "World"
    return 'Salutations World'
"""
txm_nested = TXM('sample_input_code/input_code_nested.py')
# assign 'Salutations' to 'alt_greeting' variable
new_node = ast.Assign(
    targets=[ast.Name(id='alt_greeting', ctx=ast.Store())],
    value=ast.Constant(value='Salutations')
)
txm_nested.apply_insert(new_node, (0,))
# assign 'greeting' to 'alt_greeting' inside if statement
new_node = ast.Assign(
    targets=[ast.Name(id='greeting', ctx=ast.Store())],
    value=ast.Name(id='alt_greeting', ctx=ast.Load())
)
txm_nested.apply_insert(new_node, (1,0))
end_program = """def hello_world(greeting):
    alt_greeting = 'Salutations'
    if greeting == 'Hello':
        greeting = alt_greeting
        return greeting + 'World'
    return 'Salutations World'
"""
assert txm_nested.assert_program_equality(end_program)
```

Fig. 10. Insert commands in nested functions.



```python
"""sample_input_code/input_code_joinedstr.py
def hello_world(greeting):
    version = 2
    return f"{greeting} World{version:.2d}"
"""
txm_nested = TXM('sample_input_code/input_code_joinedstr.py')
# update formatting of "greeting" to STRING_FORMATTING, "version" to .2f
txm_nested.apply_update_formattedvalue("greeting", STRING_FORMATTING, None)
txm_nested.apply_update_formattedvalue("version", NO_FORMATTING, ".2f")
# add '! {eod}' to the end of the string, where 'eod' is assigned to '<over>'
new_node = ast.Assign(
    targets=[ast.Name(id='eod', ctx=ast.Store())],
    value=ast.Constant(value='<over>')
)
txm_nested.apply_insert(new_node, (1,))
new_values = [
    ast.Constant(value='! '),
    ast.FormattedValue(
        value=ast.Name(id='eod', ctx=ast.Load()),
        conversion=-1
    )
]
txm_nested.apply_addrem_joinedstr(0, new_values)
end_program = """def hello_world(greeting):
    version = 2
    eod = '<over>'
    return f'{greeting!s} World{version:.2f}! {eod}'
"""
assert txm_nested.assert_program_equality(end_program)
# remove the 'greeting' and '! {eod}' from the fstring
txm_nested.apply_addrem_joinedstr(0, [-1,-2,0])
end_program = """def hello_world(greeting):
    version = 2
    eod = '<over>'
    return f' World{version:.2f}'
"""
assert txm_nested.assert_program_equality(end_program)
```

Fig. 11. Update fstrings: formatting and content.



```python
"""sample_input_code/input_code_attribute.py
def hello_world(greeting):
    if greeting.string == 'Hello':
        return greeting.greeting + 'World'
    return 'Salutations World'
"""
txm_attribute = TXM('sample_input_code/input_code_attribute.py')
# replace the 'string' attribute of 'greeting' with 'greeting'
txm_attribute.apply_replace("string", "greeting", CONSTTOCONST)
end_program = """def hello_world(greeting):
    if greeting.greeting == 'Hello':
        return greeting.greeting + 'World'
    return 'Salutations World'
"""
assert txm_attribute.assert_program_equality(end_program)
# replace all instances of the 'greeting' object with a 'social' object
txm_attribute.apply_replace("hello_world", (None,[0],["social"]), FUNC)
txm_attribute.apply_replace("greeting", "social", VARTOVAR)
txm_attribute.apply_replace("greeting", "social", VARTOVAR)
end_program = """def hello_world(social):
    if social.greeting == 'Hello':
        return social.greeting + 'World'
    return 'Salutations World'
"""
assert txm_attribute.assert_program_equality(end_program)
```

Fig. 12. Update use of objects and object attributes.

15th International Conference on Mining Software Repositories (MSR), pp. 476–486, 2018.

[2] S. Panthaplackel, M. Gligoric, R. J. Mooney, and J. J. Li, “Associating natural language comment and source code entities,” in *AAAI*, pp. 8592–8599, 2020.

[3] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, “A neural architecture for generating natural language descriptions from source code changes,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, (Vancouver, Canada), pp. 287–292, Association for Computational Linguistics, July 2017.

[4] P. Loyola, E. Marrese-Taylor, J. Balazs, Y. Matsuo, and F. Satoh, “Content aware source code change description generation,” in *Proceedings of the 11th International Conference on Natural Language Generation*, (Tilburg University, The Netherlands), pp. 119–128, Association for Computational Linguistics, Nov. 2018.

[5] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, p. 135–146, IEEE Press, 2017.

[6] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. Mooney, “Learning to update natural language comments based on code changes,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 1853–1868, Association for Computational Linguistics, July 2020.

[7] R. J. Kate, Y. W. Wong, and R. J. Mooney, “Learning to transform natural to formal languages,” in *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI’05, p. 1062–1068, AAAI Press, 2005.

[8] J. Krishnamurthy, P. Dasigi, and M. Gardner, “Neural semantic parsing with type constraints for semi-structured tables,” in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 1516–1526, Association for Computational Linguistics, Sept. 2017.

[9] D. L. Waltz, “An english language question answering system for a large relational database,” *Commun. ACM*, vol. 21, p. 526–539, July 1978.

[10] W.-t. Yih, X. He, and C. Meek, “Semantic parsing for single-relation question answering,” in *Proceedings*

of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), (Baltimore, Maryland), pp. 643–648, Association for Computational Linguistics, June 2014.

[11] P. Liang, M. Jordan, and D. Klein, "Learning Dependency-Based Compositional Semantics," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, (Portland, Oregon, USA), pp. 590–599, Association for Computational Linguistics, June 2011.

[12] J. Andreas, A. Vlachos, and S. Clark, "Semantic parsing as machine translation," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, (Sofia, Bulgaria), pp. 47–52, Association for Computational Linguistics, Aug. 2013.

[13] Y. W. Wong and R. Mooney, "Learning for semantic parsing with statistical machine translation," in *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, (New York City, USA), pp. 439–446, Association for Computational Linguistics, June 2006.

[14] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 440–450, Association for Computational Linguistics, July 2017.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1643–1652, Association for Computational Linguistics, Oct.-Nov. 2018.

[16] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Berlin, Germany), pp. 599–609, Association for Computational Linguistics, Aug. 2016.

[17] E. C. R. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, "Program synthesis and semantic parsing with learned code idioms," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada* (H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, eds.), pp. 10824–10834, 2019.

[18] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," 2017.

[19] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," AAAI'96, p. 1050–1055, AAAI Press, 1996.

[20] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *EMNLP*, 2018.

[21] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Berlin, Germany), pp. 33–43, Association for Computational Linguistics, Aug. 2016.

[22] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 1139–1149, Association for Computational Linguistics, July 2017.

[23] T. Johnson, "Natural language computing: The commercial applications," *The Knowledge Engineering Review*, vol. 1, no. 3, p. 11–23, 1984.

[24] W. A. Woods, "Progress in natural language understanding: An application to lunar geology," in *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, AFIPS '73, (New York, NY, USA), p. 441–450, Association for Computing Machinery, 1973.

[25] P. C. Lockemann and F. B. Thompson, "A rapidly extensible language system (the REL language processor)," in *International Conference on Computational Linguistics COLING 1969: Preprint No. 34*, (Snga Säby, Sweden), Sept. 1969.

[26] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a natural language interface to complex data," *ACM Trans. Database Syst.*, vol. 3, p. 105–147, June 1978.

[27] Y. Artzi and L. Zettlemoyer, "Weakly supervised learning of semantic parsers for mapping instructions to actions," *Transactions of the Association for Computational Linguistics*, vol. 1, pp. 49–62, 2013.

[28] L. S. Zettlemoyer and M. Collins, "Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars," UAI'05, (Arlington, Virginia, USA), p. 658–666, AUAI Press, 2005.

[29] L. Zettlemoyer and M. Collins, "Online learning of relaxed CCG grammars for parsing to logical form," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, (Prague, Czech Republic), pp. 678–687, Association for Computational Linguistics, June 2007.

[30] Y. Artzi and L. Zettlemoyer, "Bootstrapping semantic parsers from conversations," in *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, (Edinburgh, Scotland, UK.), pp. 421–432, Association for Computational Linguistics, July 2011.

[31] Y. Artzi and L. S. Zettlemoyer, *Situated understanding and learning of natural language*. PhD thesis, 2015.

[32] Y. Artzi, N. Fitzgerald, and L. Zettlemoyer, "Semantic parsing with Combinatory Categorial Grammars," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, (Doha, Qatar), Association for Computational Linguistics, Oct. 2014.

[33] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Melbourne, Australia), pp. 731–742, Association for Computational Linguistics, July 2018.

[34] M. Nye, L. Hewitt, J. Tenenbaum, and A. Solar-Lezama, "Learning to infer program sketches," 09–15 Jun 2019.

[35] Semantic Machines, J. Andreas, J. Bufe, D. Burkett, C. Chen, J. Clausman, J. Crawford, K. Crim, J. DeLoach, L. Dorner, J. Eisner, H. Fang, A. Guo, D. Hall, K. Hayes, K. Hill, D. Ho, W. Iwaszuk, S. Jha, D. Klein, J. Krishnamurthy, T. Lanman, P. Liang, C. H. Lin, I. Lintsbakh, A. McGovern, A. Nisnevich, A. Pauls, D. Petters, B. Read, D. Roth, S. Roy, J. Rusak, B. Short, D. Slomin, B. Snyder, S. Striplin, Y. Su, Z. Tellman, S. Thomson, A. Vorobev, I. Witoszko, J. Wolfe, A. Wray, Y. Zhang, and A. Zotov, "Task-oriented dialogue as dataflow synthesis," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 556–571, Sept. 2020.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[37] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (Online), pp. 2655–2668, Association for Computational Linguistics, June 2021.

[38] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 7871–7880, Association for Computational Linguistics, July 2020.

[39] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 150–162, 2021.

[40] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[41] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[42] I. Solaiman, M. Brundage, J. Clark, A. Askell, A. Herbert-Voss, J. Wu, A. Radford, and J. Wang, "Release strategies and the social impacts of language models," *ArXiv*, vol. abs/1908.09203, 2019.

[43] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever,

and D. Amodei, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

[44] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021.

[45] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, p. 56–65, Dec 2019.

[46] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE ’20, (New York, NY, USA), p. 602–614, Association for Computing Machinery, 2020.

[47] J. Devlin, J. Uesato, R. Singh, and P. Kohli, “Semantic code repair using neuro-symbolic transformation networks,” *ArXiv*, vol. abs/1710.11054, 2017.

[48] M. Yasunaga and P. Liang, “Graph-based, self-supervised program repair from diagnostic feedback,” in *ICML*, pp. 10799–10808, 2020.

[49] B. Berabi, J. He, V. Raychev, and M. Vechev, “Tfix: Learning to fix coding errors with a text-to-text transformer,” in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 780–791, PMLR, 18–24 Jul 2021.

[50]

[51] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. Mooney, “Learning to update natural language comments based on code changes,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 1853–1868, Association for Computational Linguistics, July 2020.