Program Generation from Natural Languages

Allan Wang and **Youngsun Jin**
McGill University
{allan.wang,youngsun.jin}@mail.mcgill.ca

Abstract

Most programming languages help developers learn by providing examples through a list of common executions. While useful and detailed, sifting through examples to find a particular use case can be time consuming. If we can instead provide examples by matching simple input instructions in English, we can help refine the amount of code a developer has to go through, and assist in creating an infinite set of examples tailored to each user. In this work, we develop a prototype system which is able to process natural language instructions related to list creation with conditions, and translate them into executable code for multiple programming languages.

1 Introduction

Learning the syntax of a programming language can be frustrating and challenging. For a new developer, programming languages contain lots of domain-specific rules, varying in control structures, syntax, etc. For experienced developers, programming languages may introduce completely new paradigms, and may not be easily translated from known languages through one to one mappings of keywords. One way to help overcome this technical hurdle is to read official documentation, or a large set of common examples. However, this becomes a big time investment for a user who just wants to explore the basics of a language. Moreover, the examples may not completely reflect the desired use case, and can often be irrelevant to the problem at hand.

Our goal is to explore the feasibility of mapping natural languages to programs by extracting features and creating runnable code. This would greatly reduce the learning curve when dealing with simple operations, and will help introduce language features without the developer knowing about them beforehand. More specifically, we wish to focus on generating number lists in multiple programming languages, allowing for varying constraints available across all supported languages. We will use Combinatory Categorial Grammar (CCG) to create a tree representing the logic, then introduce language specific transformers to map subtrees to working code. Lastly, we will pass through the generated body and output a working program. On top of being able to create a program, we wish to introduce a structure that can be extended to support both new languages and new operations within the semantics tree.

2 Related Work

2.1 Natural Language to Database Query

One of the earliest system that makes use of natural language translation into executable code is a Natural Language Interface for a Database (Androutsopoulos et al., 1995). When databases contain enormous number records, a specific query language such as the Structured Query Language (SQL) needs to be used in order to extract records specified by the user. This means that one needs to understand the syntax for querying the database and the way to join the underlying tables to extract necessary records. However, this introduces difficulties for users because they may not be aware of the structure of the database. A NLIDB simplifies the task by providing the user interface to input natural languages to query the database. We wish to accomplish a similar task, but in the scope of general program code vs SQL. For feasibility, we have simplified our problem to creating a list of numbers with conditions that can be applied to each number inside the list.

2.2 Natural Language to Java

NaturalJava (Price et al., 2000) is one of the early systems that allows to create Java programs from English commands. The system accepts English sentences as input and then uses information extraction techniques to map one of 400 case frames. The interpreter infers changes that have to be made using a decision tree. In response, the Abstract Syntax Tree (AST) is modified, and it is automatically converted into Java source code. Although our task is similar in a sense that both translate natural language into code, we do not manually create case frames to infer programs. Our task is more involved with parsing the natural language sentence using a set of rules to generate semantic representation of the input.

2.3 Unrestricted Natural Language

One of the most powerful natural language interface was developed (Vadas and Curran, 2005). They presented the prototype system which takes unrestricted English as input and output code in the Python programming language. They used Combinatory Categorial Grammar (CCG) as a base tool to generate syntactic categories for English sentences. Then the *ccg2sem* system (Blackburn and Bos, 2005) was used to convert CCG parse trees into Discourse Representation Structure (DRS). Having extracted the functional verb and its arguments, if the semantic information matches with predefined primitives, then the equivalent code is generated. Unlike our task, they trained the new model for the parser on the manually-generated corpus as a result of the differences between standard English sentences and programming statements. However, in this work, we predefined the set of rules in CCG to assign each word with functional feature instead of syntactic category, which will be then used to generate the semantic representation of the whole sentence.

3 Methodology

3.1 CCG Parsing

In creating our grammar, we defined a set of rules for each type we may encounter. We have the following primitives: `program`, `creation`, and `range`. A program represents an independent set of rules that is capable or defining a runnable program. A creation denotes a rule that generates new data; this can be any viable programming type, but in our case specifically, it represents an integer list. A range represents a phrase with a minimum number and maximum number; this is necessary in creating a proper list.

For conditions, we can create higher order types that build upon existing lists. For instance, an ‘even list from 0 to 100’ would still represent the `list` primitive, with a new feature attached. Here, we begin to face some problems. We can for instance want a ‘list from 0 to 100 that is divisible by 4’, but a ‘divisible by 4 list from 0 to 100’ would not make sense. We can work around this by introducing special groups per condition, and making sure that our program rules respect the expected groups based on the order of the initial sentence.

Another problem we face is supporting number parsing, where a number primitive can be any valid sequence of digits. One way we considered was post processing, where we would replace each number with a special symbol, create our tree, then add back the numbers in the same order as they appeared in the sentence. This poses a problem, as our assumption of word order may not always be correct. As an example, both ‘create a list from 0 to 100’ and ‘create a list to 100 from 0’ are represented by the same tree, and yet the input numbers are not in the same order. We worked around this by extracting all the numbers in the word token, adding rules that map them to the same integer primitive, and then creating a new CCG parser for each sentence.

The final result allows us to create CCGs like in figure 1.

3.2 Language Formatting

As we wish to parse our input using a CCG, we must first ensure that our corpus is contained in our lexicon. We can simply do so by removing all words that are not part of our lexicon, but that would greatly

```
            list                  from         0         to          100
((L/RF)/RT) {\y x.list(x,y)}  (RF/I) {\x.x}  I {0}  (RT/I) {\x.x}  I {100}
                              ---------------------->
                                      RF {0}
                              ----------------------<T
                              (L\(L/RF)) {\F.F(0)}
----------------------------------------------------<B
              (L/RT) {\y.list(0,y)}
                                                    ------------------------>
                                                            RT {100}
---------------------------------------------------------------------------->
                              L {list(0,100)}
```

Figure 1: Shortened example of CCG output.

constrain our input, and may also result in code that does not represent what we wanted. To support a wider set of tokens, we preprocess the text with a dictionary of token words to NLTK lemma keys. We can use the keys to extract synonyms, and, provided that each word is unique to a specific token, we can replace them without altering the semantics.

We still need to filter out the remaining words, but in checking the part of speeches, we noted that condition features typically contain an adjective or adverb. We can therefore filter out words without those part of speeches, and return an error when other unknown tokens are found. This allows us to parse instructions beyond the keywords that we select, and allows us to reject inputs that we cannot parse with reasonable confidence.

3.3 Feature Mapping

Once we have our semantics tree, we can map them into programming rules. By the way we defined our primitives, it is noted that each node can lead to a part of a programming with the information within its subtree. As we are familiar with both the number of children per primitive, as well as their orders, we can create code templates that take in child subcomponents and generate a new subcomponent. Once we get to the program primitive, the result should be a working program.

3.4 Code Generation

Given that our feature mapping aims to convert a semantics tree to code, our code generation is essentially applying our feature map to the root node. However, the process is not so simple. A common part of many programming languages is a set of import rules at the top of the file. While we can add each rule in our template if they don't appear elsewhere in the subtree, this can quickly become messy for large trees. A simpler method is to aggregate all necessary imports, and to post process our code to ensure we add unique imports. We did so by introducing a variable in our feature map output, so that each token is mapped to both a template and an import set.

It is worth noting that this method is quite extensible, as it does not affect the generation of our program body. We can almost view it as a new feature tree, where we can define a new mapping to further convert it into code. In the case of imports, the mapping per node is to simply aggregate the unique key set and to output each of them in a new line.

3.5 Negation Extension

To test the extensibility of our generator, we implemented negation using our existing structure. We may note that a negation can take place before any existing condition, resulting in a new predicate with the opposite output. Furthermore, as an optimization, two consecutive negations cancel out. While we could preprocess the text and remove consecutive negations, we decided to handle this within the parse tree. To support the negation feature, we add another attribute in our feature map for conditions only. When we encounter a negation feature in our tree, we simply take the child and flip the negation field. Note that this does mean that we have to ap-

```
fun code() =
      (0..100).asSequence()
              .filter { it % 2 == 0 }
              .filter { it > 5 }
              .toList()
```

Figure 2: Kotlin program equivalent to 'create even list from 0 to 100 that is bigger than 5'

ply our template function lazily, as we can only be sure of a condition's negation after the full sentence is parsed.

4 Result

Our baseline is the nearest-neighbour approach. Given a table of program-description pairs and the input of a natural language sentence, we search for the closest description in a table of program-description pairs using cosine similarity, and return the corresponding line of code.

The baseline has a major flaw. In order to compare two descriptions of code, we first needed to convert text into bag-of-words features. Bag-of-words have two major weaknesses: they lose the ordering of words and the semantics of words are neglected. In our task, the order of words cannot be ignored because we need to know which conditions are being negated. Furthermore, bag-of-words features cannot interpret the semantics of double negation. In other words, the baseline looks for the sentence that has nearly the identical number of the negation term, regardless of whether the cancellation occurs or not.

This problem is overcome by the use of CCG, by tagging each word as a functional feature and using inference rules to derive the semantic representation of natural language sentences. In regards to the evaluation of our model, unfortunately we could not find any test corpus that is appropriate for our task. Instead, we have manually constructed the testing instances in the unit test environment. Each test instance passes if the output of the generated code matches the expected output. We successfully covered 10 testing instances with 4 different programming languages, namely Python, Kotlin, Java, and Elm. Figures 2 and 3 show examples of code output in Kotlin and Java respectively.

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

class Code {
  public static List<Integer> code() {
    return IntStream.rangeClosed(0, 100)
      .filter(x -> x % 2 == 0)
      .filter(x -> x > 5)
      .boxed()
      .collect(Collectors.toList());
  }
}
```

Figure 3: Java program equivalent to 'create even list from 0 to 100 that is bigger than 5'

5 Conclusion

From our results, we show that it is possible to create a program generator, whose complexity scales relatively linearly with the problem domain. We have successfully shown that it is easy to extend upon the generator with minimal changes on existing logic, and that it is easy to support new languages.

The project is currently limited in that we focused exclusively on numbered lists, with operations that work primarily on integers. An extension can be made to allow for differing types, and also to support more mutation oriented programming versus pure functional programming. While we have shown that it is feasible to embed features that aren't in a language's stdlib (such as checking for prime numbers in most programming languages), we have not fully explored the use case. Lastly, our code generation does not give the user the ability to name their variables. This is primarily due to our idea that variable naming is not a typical restriction provided by non programmers, but it may be interesting to see how the context of the natural language inputs can allow for such changes.

References

Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1:29–81.

Patrick Blackburn and Johan Bos. 2005. Representation and inference for natural language: A first course in computational semantics. *Computational Linguistics*, 32:283–286.

Johan Bos, Stephen Clark, Mark Steedman, James R. Curran, and Julia Hockenmaier. 2004. Wide-coverage semantic representations from a ccg parser. In *COLING*.

David Price, Ellen Riloff, Joseph L. Zachary, and Brandon Harvey. 2000. Naturaljava: a natural language interface for programming in java. In *IUI*.

Chris Quirk, Raymond J. Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*.

David Vadas and James R. Curran. 2005. Programming with unrestricted natural language. In *ALTA*.

6 Statement of Contribution

Youngsun Jin partially implemented the CCG parser, found external references, and compared the results with a baseline.

Allan Wang partially implemented the CCG parser, wrote the code generator, and wrote the methodology.