Code Generation for Collectible Card Games with Complex APIs

John Licato,[1] **Logan Fields,**[1] **Brayden Hollis**[2]

[1] Advancing Machine and Human Reasoning (AMHR) Lab
Department of Computer Science and Engineering
University of South Florida

[2] Information Directorate
Air Force Research Lab
licato@usf.edu, ldfields@usf.edu, brayden.hollis.1@us.af.mil

Abstract

Large pre-trained language models (LMs) such as GPT-3 Codex are able to generate code remarkably well given prompts of natural language text. But if we want to use such LMs to generate code compatible with a specific API or library (e.g., an API which provides the environments in which certain rules, laws, or orders are to be carried out), the amount of computational and data resources required to fine-tune such models can be cost prohibitive to most organizations. Given these practical limitations, is it possible to utilize these massive code-generation LMs to write code compatible with a given API? We develop an algorithm that selects code examples using a smaller LM trained to predict which features of an API are *likely* to be used in the resulting code, which is a simpler problem than actually generating the code. The selected examples are then used to build a prompt for the larger LM, which in turn generates the final code. We demonstrate our results on a benchmark dataset derived from the collectible card game "Magic: the Gathering," and obtain state-of-the-art results.

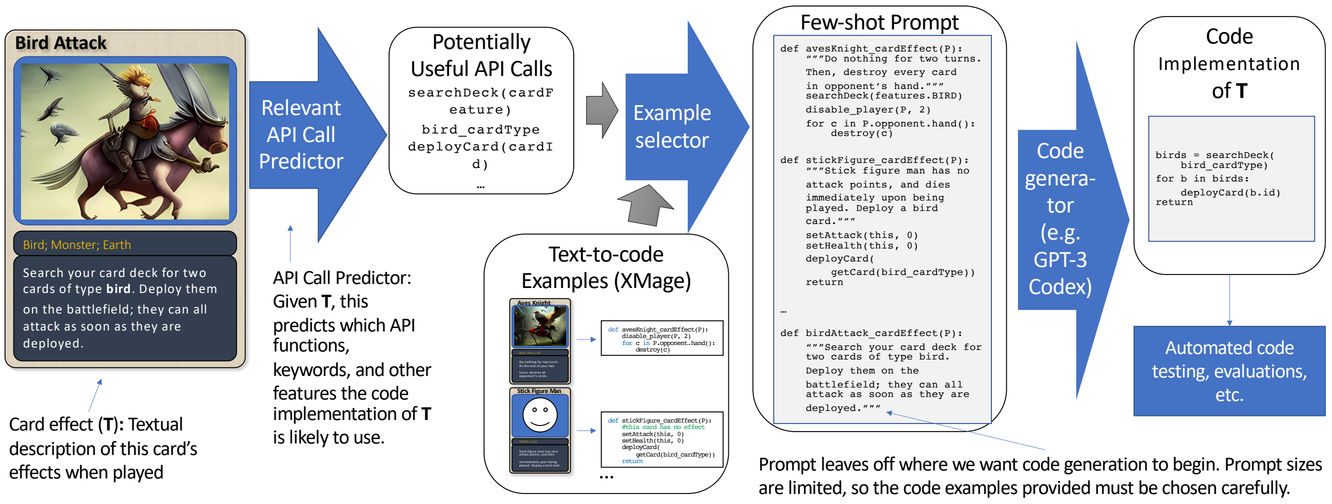
Introduction

The problem of how to automatically interpret language in a way that produces a scrutable, understandable interpretation is of fundamental importance for the future of AI. It has been argued, for instance, that in order for artificially intelligent systems to properly follow human laws, they need to be able to interpret them, which requires resolving the sometimes-ambiguous language that laws use. But furthermore, the interpretation the AI chooses must be provided in a form that stakeholders can inspect, test, and use as precedent for future interpretations (Licato 2022a; 2022b; 2021). In other words, given text to be interpreted by an AI, human stakeholders need to be able to inspect: (I1) *how* the AI interpreted that text, and (I2) *why* the AI believes that interpretation is best. An emerging body of work is exploring approaches to (I2) under the topic of *interpretive reasoning* (Licato 2021; Sartor et al. 2014; Walton, Sartor, and Macagno 2018; Licato, Marji, and Abraham 2019; Walton, Macagno, and Sartor 2021; Araszkiewicz 2021), but in this paper we will focus on (I1).

One paradigm to productively study (I1) is that of *automatic code generation*, particularly when the code is generated from law-like text. Consider, for example, collectible card games (CCGs) such as *Magic: the Gathering*™(MtG). In such games, players have a set of cards they can play, each of which may have text describing possible effects of those cards. Some of these effects might be simple and unambiguous (e.g., "When this card is played, the opposing player takes 5 damage."). But it is very common for cards to contain language that is *open-textured*—i.e., language whose full domain of applicability is underspecified, often by design (Hart 1961; Waismann 1965; Vecht 2020). Correctly interpreting open-textured language in CCGs can require a deep knowledge of the game, how the card and the language used in the text has historically been interpreted, the ramifications of adopting one interpretation over another, and so on. These problems parallel those faced by the automatic interpretation of law (Bench-Capon and Sergot 1985; Sanders 1991; Franklin 2012; Prakken 2017; Quandt and Licato 2020); in fact, the rules and conflict resolution systems of MtG has been described as an entire legal system in and of itself, complete with judges who must make rulings based on precedent, established best practices for interpretive reasoning, and commonsense notions of fairness (Adler 2019). As such, the problem of how to automatically translate the text on MtG cards (and CCGs in general) into programming language code is a productive way to tackle the larger problem of automated reasoning about legal text, since converting the text into code forces one to make decisions about how the text is to be interpreted.

Background

Automatic Code Generation Generating programming code from a mixed input of natural language text and structured prompts is a difficult challenge even for state-of-the-art language models. Previous models for code generation largely focused on syntactically matching gold-standard code, without regard for functionality. This created a large oversight wherein such models were not able to innovatively generate code that is syntactically different, but functionally equivalent to the gold standard. This is particularly salient for the generation of code from CCG card effects, as many keywords may produce similar effects with varying magnitudes and differing game-specific keywords across



Bird Attack
Bird; Monster; Earth
Search your card deck for two cards of type **bird**. Deploy them on the battlefield; they can all attack as soon as they are deployed.
Card effect (**T**): Textual description of this card's effects when played
Relevant API Call Predictor
API Call Predictor: Given **T**, this predicts which API functions, keywords, and other features the code implementation of **T** is likely to use.
Potentially Useful API Calls
searchDeck(cardFeature)
bird_cardType
deployCard(cardId)
…
Example selector
Text-to-code Examples (XMage)
Aves Knight
Bird; Hero; Air
Do nothing for two turns. At the end of your two turns, destroy all opponent's cards.
Stick Figure Man
Villain; Lazy
Stick figure man has zero attack points, and dies immediately upon being played. Deploy a bird card.
…
Few-shot Prompt
Code generator (e.g. GPT-3 Codex)
Code Implementation of **T**
Automated code testing, evaluations, etc.
Prompt leaves off where we want code generation to begin. Prompt sizes are limited, so the code examples provided must be chosen carefully.

```
def avesKnight_cardEffect(P):
    disable_player(P, 2)
    for c in P.opponent.hand():
        destroy(c)
```

```
def stickFigure_cardEffect(P):
    #this card has no effect
    setAttack(this, 0)
    setHealth(this, 0)
    deployCard(
        getCard(bird_cardType))
    return
```

```
def avesKnight_cardEffect(P):
    """Do nothing for two turns.
    Then, destroy every card
    in opponent's hand."""
    searchDeck(features.BIRD)
    disable_player(P, 2)
    for c in P.opponent.hand():
        destroy(c)

def stickFigure_cardEffect(P):
    """Stick figure man has no
    attack points, and dies
    immediately upon being
    played. Deploy a bird
    card."""
    setAttack(this, 0)
    setHealth(this, 0)
    deployCard(
        getCard(bird_cardType))
    return

…

def birdAttack_cardEffect(P):
    """Search your card deck for
    two cards of type bird.
    Deploy them on the
    battlefield; they can all
    attack as soon as they are
    deployed."""
```

```
birds = searchDeck(
    bird_cardType)
for b in birds:
    deployCard(b.id)
return
```

Figure 1: Our process fine-tunes a smaller language model to predict which API calls might be used, and then select example code snippets based on its predictions. Those code snippets are then placed into a few-shot prompt that the code generator uses to write code.

CCGs may produce the equivalent or near-equivalent in-game effects. Evaluation metrics such as CodeBLEU (Ren et al. 2020) attempt to address the functionality problem by weighting the BLEU score for assessing multiple "translations" along with the syntactic abstract syntax tree match, but this still prefers predictions that are formatted similarly to the gold standard and underperforms human judgment on choosing effective code (Evtikhiev et al. 2022). The challenge of generating code for card effects in CCGs is further exacerbated due to fact that many cards may contain "flavor text," e.g., text that describes how the character represented in a card relates to the larger mythology of the game, but does not modify the card's effects in any way.

Ling et al. used latent predictor networks to automatically generate code for MtG and Blizzard Entertainment's *Hearthstone* (Grad 2017) cards (Ling et al. 2016). Their method was able to identify common effects and generate code for cards in the test set, which were then evaluated for their similarity to others in the train set. However, the model was unable to generalize natural language terms outside the scope of game-specific keywords and, therefore, produced high inaccuracies when presented with card effects it had not previously seen. Further, they did not address functional correctness, as generated code that was functionally equivalent but syntactically different from correct code may have been marked as incorrect.

Many current models have focused on addressing the functionality concern (Zhong, Yu, and Klein 2020; Rajkumar, Li, and Bahdanau 2022). The CodeGen-Test model (Zhong et al. 2022) implements a code testing step in the model training and backpropagates the results of the tested code along with the syntactic similarity metrics. The model was found to outperform the existing state-of-the-art models in similarity as measured by BLEU, which performs poorly in differentiating code, and Rouge-L, which is better at emulating human judgment (Evtikhiev et al. 2022). Furthermore, it also significantly outperformed those same models in functional correctness.

With the massive growth of large language models (LMs), even the process of fine-tuning has become computationally expensive to the point of infeasibility for many tasks. However, methods to teach these models to perform specific tasks are still necessary. The emergence of massive generative language models like GPT-3 (Brown et al. 2020) has led to an increased interest in the use of *prompt engineering*, whereby the researcher focuses on how to structure the prompts that are given to the LMs to generate their outputs, rather than on fine-tuning the LM. Prompts can also be fine-tuned by feeding them into language models and adjusting the prompt based on the output received.

OpenAI's GPT-3 Codex (Chen et al. 2021) was trained on code from Github and performed at state-of-the-art levels in code generation tasks. Codex produced significantly improved results on HumanEval for measuring functional correctness, compared to GPT-3 and GPT-J. Subsequent models which use prompting to build on top of Codex and generate SQL code have shown further improvements in fluency (Scholak, Schucher, and Bahdanau 2021; Poesia et al. 2022) and adequacy (Rajkumar, Li, and Bahdanau 2022; Trummer 2022). Codex is able to take, as its input prompt, code examples or human-understandable text describing code to be written.

Presumably, we can use the prompt to tell Codex to write the code implementation of a card in a CCG, given the text written on the card. The prompt can be quite large: the size of the prompt plus the maximum output size is roughly 8,000 tokens (each token is roughly 3 characters). But even this prompt size is not large enough to teach Codex how to use the massive XMage API. And fine-tuning Codex is simply not plausible for multiple reasons: GPT-3 is not publicly released, and even if it were, it requires massive computational capabilities far beyond that of most organizations (including most research Universities). In this paper, we propose a prompt engineering method for breaking this impasse without resorting to fine-tuning GPT-3.

Prior Work in the Intersection of AI and CCGs Given their intricate natural language effects and expansive action spaces, CCGs present a number of interesting AI research questions. Deckbuilding, for example, is the problem of determining which cards to place in a player's deck, given a much larger set of possible cards. CCGs typically require players to build their own card decks, either through iterative drafting of specified options or synergistically from the space of cards that the player personally owns, and then randomly draw cards from their deck to engage in spellcasting and combat. This creates a massive branching set that simple heuristic methods are unable to efficiently span, made more complex when the problem is made adversarial—i.e., how might a deck optimally be chosen to maximize a given player's strengths or capitalize on an opponent's weaknesses? Prior AI-based research into deckbuilding has shown that, although complex heuristics can outperform random guessing in ranking cards against one another (Ward et al. 2021), evolutionary algorithms (García-Sánchez et al. 2018; Kowalski and Miernik 2020; Zhang et al. 2022) currently produce the most efficient and significant performance increases.

Actually creating an AI to play a CCG presents another set of unique challenges, e.g.: choosing a play style, identifying patterns in the opponent's actions, and scaling based on difficulty (Hoover et al. 2020). Furthermore, current best-known methods of approaching these challenges vary widely based on the game; simple tree searches have shown large improvements in AI performance on LoCM (Klasiński, Meller, and Witkowski 2020; Miernik and Kowalski 2022), but more complex games like Hearthstone require neural networks (Grad 2017) to assist player performance. Some CCGs, such as Hearthstone and *Legends of Code and Magic* (LoCM) (Kowalski and Miernik 2018), have been designed to play in an entirely digital form (thus reducing some of the logistics required to develop an AI), but these have significantly smaller spaces of card effects when compared to older physical card games such as MtG and *Yu-Gi-Oh!*. Attempts have been made to digitize MtG via the open-source XMage client, but XMage implementation is large and not straightforward to automate: every single card type has at least one Java class, every possible effect has at least one function call, etc. Despite these complexities, since the benchmark dataset established by (Ling et al. 2016) uses XMage as its implementation, and this is the only existing benchmark dataset for MtG code generation, we will use it as our target as well for this paper's work.

Our Contribution To summarize the background literature: automatic generation of code from card text is a difficult problem to solve, in part because the APIs used to implement CCGs often have too many features to learn without fine-tuning. However, fine-tuning state-of-the-art code generating language models is not computationally feasible to most. We therefore propose the following solution: we will instead fine-tune a smaller LM (RoBERTa-Large) not to directly write the code to implement a card's text, but to predict *which API features the code is likely to use* (Stage 1). These predictions are then used to select examples of previously written code to build a prompt for the larger LM (GPT-3 Codex). The intuition behind this is that the larger LM simply needs examples of how to use the API features it is likely to use. The LM then writes the code (Stage 2). The full pipeline is pictured in Figure 1. To our knowledge, we are the first to propose such an approach, and we report results on the Card2Code XMage benchmark (Ling et al. 2016) beyond existing state-of-the-art.

Stage 1: Predicting API Identifiers

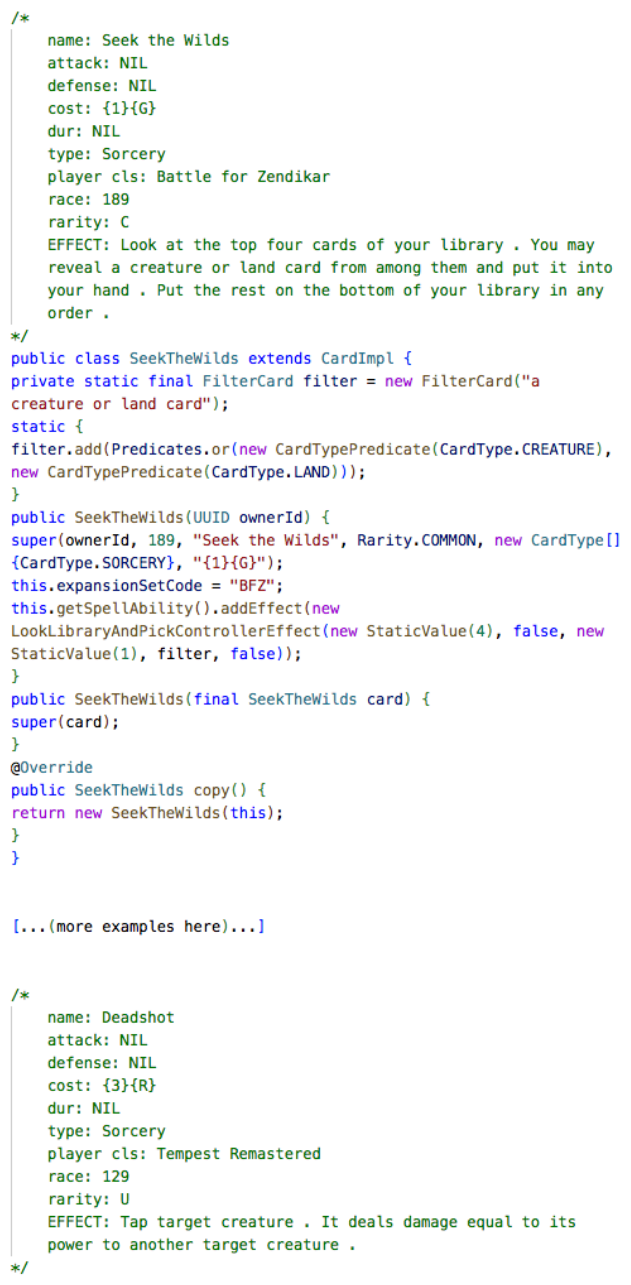
The goal of Stage 1 is to train a classifier to predict, given the text of a card, which API-specific features the code implementation of that text in the API is likely to use. Let us define *interface symbols* (ISes) as the set of symbols that are specific to a given API: class names, enumerated types, function names, etc. For the remainder of this paper, we will be referring to the ISes of the XMage API, which was written in Java. The set of ISes does not include strings, literals, or language-specific keywords (`class`, `if`, `else`, etc.). If our classifier were able to determine which ISes would be needed, we could select example code to include in the prompt that demonstrates how to use the ISes properly.

Dataset construction. Using the code as part of the XMage training set, we extracted a list of ISes, resulting in 2,712 unique symbols. Note that this large number of unique symbols is so large, that fully teaching an LM how to use the API cannot be reasonably done without significant fine-tuning, which—for state-of-the-art code generation models like GPT-3 Codex—is computationally infeasible for most organizations. Given our list of ISes, we constructed a *IS dataset* consisting of triplets (T, k, p), where:

- T is the text of some card in the XMage training set
- k is an IS
- $p \in \{0, 1\}$ is 1 if the code implementation of T contains k, 0 otherwise

For each (T, k, p) where $p = 1$, we randomly selected a k' such that $(T, k', 0)$ could be inserted into the dataset, thus keeping the dataset balanced. The resulting dataset was divided into a training set with over 303K items, and dev and test sets each with over 16K items. We then trained RoBERTa-Large on the dataset to predict p given (T, k), and achieved over 90% accuracy. The trained model could then be used to predict which keywords appear in the text implementation for any given text T by giving it (T, k) as input for all k and predicting the set of words k for which the model returned a value of 1.

Baselines. In order to evaluate whether our trained model was able to predict ISes better than random models, we created multiple baselines. The hyperparameters in the base-



```
/*
    name: Seek the Wilds
    attack: NIL
    defense: NIL
    cost: {1}{G}
    dur: NIL
    type: Sorcery
    player cls: Battle for Zendikar
    race: 189
    rarity: C
    EFFECT: Look at the top four cards of your library . You may
    reveal a creature or land card from among them and put it into
    your hand . Put the rest on the bottom of your library in any
    order .
*/
public class SeekTheWilds extends CardImpl {
private static final FilterCard filter = new FilterCard("a
creature or land card");
static {
filter.add(Predicates.or(new CardTypePredicate(CardType.CREATURE),
new CardTypePredicate(CardType.LAND)));
}
public SeekTheWilds(UUID ownerId) {
super(ownerId, 189, "Seek the Wilds", Rarity.COMMON, new CardType[]
{CardType.SORCERY}, "{1}{G}");
this.expansionSetCode = "BFZ";
this.getSpellAbility().addEffect(new
LookLibraryAndPickControllerEffect(new StaticValue(4), false, new
StaticValue(1), filter, false));
}
public SeekTheWilds(final SeekTheWilds card) {
super(card);
}
@Override
public SeekTheWilds copy() {
return new SeekTheWilds(this);
}
}


[...(more examples here)...]


/*
    name: Deadshot
    attack: NIL
    defense: NIL
    cost: {3}{R}
    dur: NIL
    type: Sorcery
    player cls: Tempest Remastered
    race: 129
    rarity: U
    EFFECT: Tap target creature . It deals damage equal to its
    power to another target creature .
*/
```

Figure 2: Example of a prompt we construct and give to GPT-3, with relevant example text/code pairs followed by the text for the code to be generated.

lines that follow were chosen through grid search, and for space we only report values that are of interest:

- Pure Random (PR): For each IS, we include that IS with $p\%$ probability ($p \in \{10, 50, 90\}$).
- Random by Frequency (RF): We analyze the XMage training set, and determine the document frequency of each IS (the percentage of code examples in which the IS appears). For each IS, its document frequency is then multiplied by $\lambda \in \{3, 5, 10\}$, and the resulting value is the probability with which we include the IS. The parameter size of $\lambda = 5$ was chosen as it made the average number of ISes per text input almost the same as that of RoBERTa-Large.
- Simple Similarity (SS): To examine the possibility that RoBERTa-Large is simply learning to select ISes that have similarities to words in the original card text, this SS model predicts an IS if is either a strict substring of, or has as a substring, a word in the card text.

	Jaccard Similarity	**Average F$_1$**
PR ($p = 10\%$)	0.49%	0.97%
PR ($p = 50\%$)	0.54%	1.06%
PR ($p = 90\%$)	0.53%	1.05%
RF ($\lambda = 3$)	3.21%	6.05%
RF ($\lambda = 5$)	3.68%	6.89%
RF ($\lambda = 10$)	3.53%	6.64%
SS	0.56%	1.11%
RoBERTa-Large	**18.78%**	**28.13%**

Table 1: Results of Training a Classifier to Predict ISes

Results. Because the number of ISes that actually appear in each card's code is very low compared to the number of possible ISes (roughly 45 versus 2,712), any model that predicts that no ISes should be included would be given a high accuracy, meaning it is not a useful measure here. We instead report two measures: Jaccard similarity and Average F$_1$ score (F$_1$ score for each instance, averaged over all instances). Results are listed in Table 1. Although there is room for improvement, our RoBERTa-Large model clearly outperforms all comparisons.

In order to determine whether RoBERTa-Large is simply learning to predict ISes that appear frequently, we compared the set of ISes predicted by RoBERTa-Large and RF ($\lambda = 5$). The average number of ISes predicted by these models were 159.3 and 136.6, respectively (compare this to the correct number of ISes which was on average 44.97), but the average about of ISes predicted by both models was only 16.45. This suggests that RoBERTa-Large is using a different heuristic to select ISes than simply IS frequency, and perhaps that a hybrid approach leveraging both approaches may yield improvements.

Stage 2: Generating Code from Predicted Identifiers

Prompt Construction. Given a model that can predict which ISes will be used in the generated code, our next step is to actually generate that code. We use GPT-3 Codex (Chen et al. 2021), which allows a total of 8,000 tokens for both the input prompt and the returned tokens (for reference, each token is roughly 3.1 characters on average). Since the largest code examples from the XMage training set were around 1500 tokens, we allocated 6500 tokens to construct a prompt that provides sufficient examples to teach Codex how to use the predicted ISes.

Given a card text T, and predicted ISes $\mathbf{IS}$, we must now select code snippets c_i, where each c_i consists of a Java comment stating the original card text that this code implements, and the Java code implementation itself (departing from the original work in (Ling et al. 2016), we rewrite card properties in a more human-friendly format, and restore line breaks into the Java code, in order to better match the human-readable comments in the corpus Codex was trained on). All c_i come directly from the XMage training set, which contains roughly 120K items. Furthermore, all of the c_i concatenated must have a total length of 6500 tokens or less. Our task can then be framed as an instance of the *weighted set cover* problem, as we must select a set of c_i where the weight $w(c_i)$ of each item is the number of tokens of c_i, and the value $v(c_1 \cup ... \cup c_n)$ is the number of $k \in (c_1 \cup ... \cup c_n)$ that are also in $\mathbf{IS}$.

Although the weighted set cover problem is known to be NP-hard, approximation algorithms exist. We use the greedy algorithm by (Chvatal 1979), which returns a set cover of weight at most $\sum_{i=1}^{m} 1/m$ times the minimum weight of the optimal cover, where m is the largest set size (in our case, the c_i with the largest number of tokens). To summarize the greedy algorithm: suppose at any iteration we have a current prompt P, consisting of the concatenation of c_i we have already chosen to include. We calculate the scores of each c_i as:

$$\frac{w(c_i)}{v(P \cup c_i) - v(P)} \text{ if } w(P \cup c_i) < 6500\text{, else } \infty$$

We then select the c_i whose score is smallest and concatenate it to P. If all scores are ∞ then we terminate. Finally, a comment is appended to P containing the text that we want GPT-3 to write the code for (see example prompt in Figure 2).

Baselines. In Stage 1, the Random by Frequency (RF) baseline with $\lambda = 5$ performed second-best to RoBERTa-Large, and the Simple Similarity (SS) baseline was the next best performer out of those using non-RF strategies. We therefore select those two baseline models to compare with RoBERTa-Large. We also use a Random Codelet (RC) baseline, which doesn't use ICs at all, and instead just randomly selects code examples from the training set until the token limit is reached. Finally, for the purposes of comparison, we introduce an *Oracle* baseline which doesn't try to predict the ISes which will appear, but actually uses the correct ISes from the XMage dev and test sets. The inclusion of the oracle is done so that we can understand how well our approach can perform if we improve the IS model from Stage 1.

The prompts from our four models are then given to GPT-3 Codex, and the generated code is compared to the correct code using four metrics: Chr-F score, a comparison of n-gram overlap which was recently recommended for use in comparing code after an analysis of similar methods (Evtikhiev et al. 2022); BLEU, typically used for comparing text similarity in language translation; CodeBLEU (Ren et al. 2020), which is an extension of BLEU score designed to compare code; and accuracy, which is true only if the generated code exactly matches the reference code.

	chr-F	**CodeBLEU**	**BLEU**	**Accuracy**
RF ($\lambda = 5$)	73.0	0.658	0.641	2.7%
SS	72.1	0.650	0.632	2.6%
RC	74.3	0.673	0.651	4.1%
RoBERTa-Large	**79.7**	**0.725**	**0.715**	**5.3%**
Oracle	86.4	0.792	0.789	9.2%

Table 2: Generated code compared to gold-standard code. The "oracle" baseline draws the ISes directly from the gold-standard code, and is included to show the upper limit of what can be achieved with perfect IS prediction.

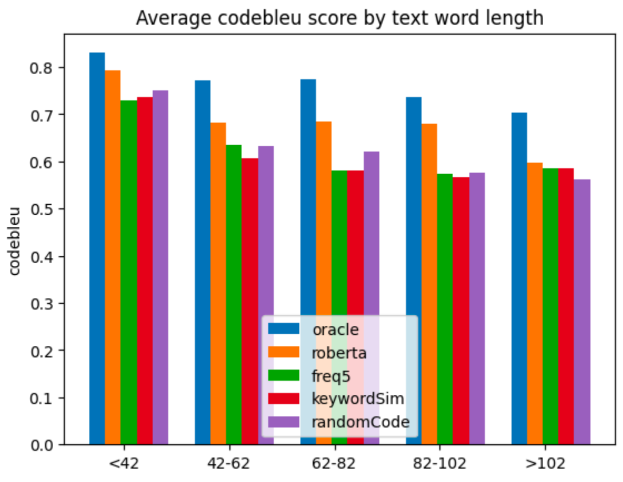
Results. Comparison of our RoBERTa-Large approach and baselines is in Table 2. Our approach outperforms all others (recall that Oracle knows in advance which ISes are correct, and is included as an upper limit), but not by a large amount. Perhaps the most informative of these four metrics is the CodeBLEU metric, as it is the only one that specifically takes into account code-specific syntactic features (the others rely on n-gram overlap or exact character match). Our results compare favorably to those reported by the best-performing models in (Ling et al. 2016), which were 0.614 (BLEU score) and 4.8% (Accuracy), but it is important to note that their results were reported on the dataset's test set, whereas ours was on its validation set (as the reported work is still considered preliminary and we wish to save the test set for future work comparing multiple code generation and prompting methods).[1]

Some cards contain effects that have only a single word (e.g., "Haste."), whereas others have lengthy descriptions with complex conditionals. To study the effect of this difference, Figure 3 shows the scores of each code generation model when broken down based on the number of words in the original card text. Although we only show these breakdowns for codebleu and accuracy for space reasons, the overall pattern is repeated across all four measures: our RoBERTa-Large model slightly outperforms or matches all other models except for Oracle, but the advantage RoBERTa-Large has over the other models disappears with larger text sizes. For card text with more than 102 words, not a single model is able to achieve full accuracy, showing that existing approaches are still quite limited.

Conclusion and Future Work

Our approach outperforms other comparisons, and elicits state-of-the-art performance from the large GPT-3 Codex LM, without requiring Codex to be fine-tuned. However, it is clear that there is much room for improvement. We suspect this can be achieved with more sophisticated dataset and training design, which we hope to carry out in next steps of this work. For example: it is our suspicion that whereas the keyword similarity-based classifier errs on the side of predicting fewer keywords (more false negatives), the RoBERTa-based classifier errs on the side of predicting

[1]Although it was our intention to report results on the test set as well, access to GPT-3 Codex was discontinued. However, we have no reason to believe that the general results reported here on the validation set would differ substantially.



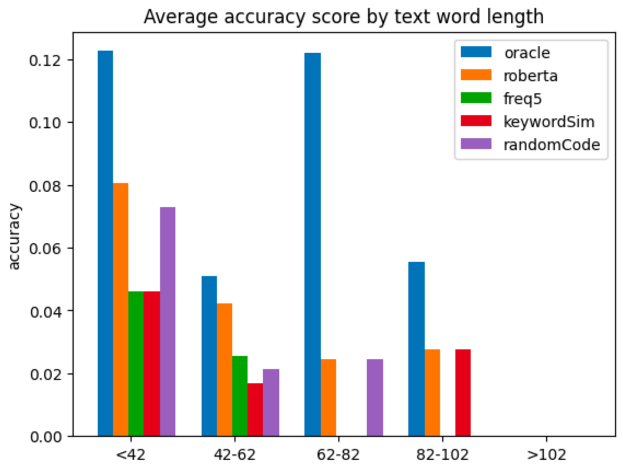


Figure 3: The discrepancy between non-oracle code generation models we tested vanishes as the word length of the original card text increases.

more keywords than needed (more false positives). It may be the case that the ideal classifier is some intelligent combination of the two.

In the pipeline we introduce in this paper, we use the prompt given to Codex to provide examples of how to write code utilizing the predicted ISes. However, instead of providing examples for use, would it be more effective to use the prompt as a way of explicitly providing instructions on how to use those ISes? At present, the XMage code base doesn't have sufficient documentation for each possible IS to make this a reasonable possibility, but this may be another productive avenue for future work.

Finally, the measures used to assess the quality of code generation are quite limited. Chr-F and BLEU do not use code-specific features, and even CodeBLEU has its limitations (Evtikhiev et al. 2022). On the other hand, the "accuracy" measure requires an exact match and does not properly evaluate alternate ways to write code that achieves the same effect. Using CCGs such as Magic: the Gathering offers a unique way to assess the quality of generated code, since the vast majority of possible card effects can be measured in terms of observable differences on the game state. Future work can take full advantage of this in order to better assess the quality of code generation.

Acknowledgments

This research was supported in part by the Air Force Research Laboratory Information Directorate, through the Air Force Office of Scientific Research Summer Faculty Fellowship Program®, Contract Numbers FA8750-15-3-6003, FA9550-15-0001 and FA9550-20-F-0005.

References

Adler, A. 2019. Keeping the law of magic: The gathering. *Escapist Magazine*.

Araszkiewicz, M. 2021. Critical questions to argumentation schemes in statutory interpretation. *Journal of Applied Logics - IfCoLog Journal of Logics and Their Applications* 8(1).

Bench-Capon, T. J. M., and Sergot, M. J. 1985. Towards a rule-based representation of open texture in law. In Walter, C., ed., *Computer Power and Legal Language: The Use of Computational Linguistics, Artificial Intelligence, and Expert Systems in the Law*.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language models are few-shot learners.

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

Chvatal, V. 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research* 4(3):233–235.

Evtikhiev, M.; Bogomolov, E.; Sokolov, Y.; and Bryksin, T. 2022. Out of the bleu: how should we assess quality of the code generation models?

Franklin, J. 2012. Discussion paper: How Much of Commonsense and Legal Reasoning is Formalizable? A Review of Conceptual Obstacles. *Law, Probability and Risk* 11(2-3):225–245.

García-Sánchez, P.; Tonda, A.; Mora, A. M.; Squillero, G.; and Merelo, J. J. 2018. Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowledge-Based Systems* 153:133–146.

Grad, L. 2017. Helping AI to play hearthstone using neural networks. In Ganzha, M.; Maciaszek, L.; and Paprzycki, M., eds., *Proceedings of the Federated Conference on Computer Science and Information Systems*, 131–134. Institute of Electrical and Electronic Engineers.

Hart, H. 1961. *The Concept of Law*. Clarendon Press.

Hoover, A. K.; Togelius, J.; Lee, S.; and de Mesentier Silva, F. 2020. The many AI challenges of hearthstone. *KI - Künstliche Intelligenz (Artificial Intelligence)* 34:33–43.

Klasiński, L.; Meller, W.; and Witkowski, M. 2020. *Implementation of Collectible Card Game AI with Opponent Prediction*. Ph.D. Dissertation, University of Wrocław.

Kowalski, J., and Miernik, R. 2018. Legends of code and magic.

Kowalski, J., and Miernik, R. 2020. Evolutionary approach to collectible arena deckbuilding using active card game genes. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. Institute of Electrical and Electronic Engineers.

Licato, J.; Marji, Z.; and Abraham, S. 2019. Scenarios and recommendations for ethical interpretive ai. In *Proceedings of the AAAI 2019 Fall Symposium on Human-Centered AI*.

Licato, J. 2021. How Should AI Interpret Rules? A Defense of Minimally Defeasible Interpretive Argumentation. *arXiv e-prints*.

Licato, J. 2022a. Automated Ethical Reasoners Must be Interpretation-Capable. In *Proceedings of the AAAI 2022 Spring Workshop on "Ethical Computing: Metrics for Measuring AI's Proficiency and Competency for Ethical Reasoning"*.

Licato, J. 2022b. War-gaming needs argument-justified ai more than explainable ai. In Sen, A., ed., *Proceedings of the XAISG Special Track on Explainable AI in Societal Games*.

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

Miernik, R., and Kowalski, J. 2022. Evolving evaluation functions for collectible card game AI. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022)*, volume 3, 253–260.

Poesia, G.; Polozov, O.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; and Gulwani, S. 2022. SYNCHROMESH: Reliable code generation from pre-trained language models.

Prakken, H. 2017. On the problem of making autonomous vehicles conform to traffic law. *Artificial Intelligence and Law* 25(3):341–363.

Quandt, R., and Licato, J. 2020. Problems of Autonomous Agents following Informal, Open-textured Rules. In Lawless, W. F.; Mittu, R.; and Sofge, D. A., eds., *Human-Machine Shared Contexts*. Academic Press.

Rajkumar, N.; Li, R.; and Bahdanau, D. 2022. Evaluating the text-to-SQL capabilities of large language models.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. Codebleu: a method for automatic evaluation of code synthesis. *CoRR* abs/2009.10297.

Sanders, K. E. 1991. Representing and reasoning about open-textured predicates. In *Proceedings of the 3rd International Conference on AI and Law (ICAIL '91)*, 137–144.

Sartor, G.; Walton, D.; Macagno, F.; and Rotolo, A. 2014. Argumentation schemes for statutory interpretation: A logical analysis. In *Legal Knowledge and Information Systems. (Proceedings of JURIX 14)*, 21–28.

Scholak, T.; Schucher, N.; and Bahdanau, D. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Moens, M.-F.; Huang, X.; Specia, L.; and tau Yih, S. W., eds., *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 9895–9901. Association for Computational Linguistics.

Trummer, I. 2022. CodexDB: synthesizing code for query processing from natural language instructions using GPT-3 codex. *Proceedings of the VLDB Endowment* 15(11):2921–2928.

Vecht, J. J. 2020. Open texture clarified. *Inquiry* 0(0):1–21.

Waismann, F. 1965. *The Principles of Linguistic Philosophy*. St. Martins Press.

Walton, D.; Macagno, F.; and Sartor, G. 2021. *Statutory Interpretation: Pragmatics and Argumentation*. Cambridge University Press.

Walton, D.; Sartor, G.; and Macagno, F. 2018. Statutory interpretation as argumentation. In Bongiovanni, G.; Postema, G.; Rotolo, A.; Sartor, G.; Valentini, C.; and Walton, D., eds., *Handbook of Legal Reasoning and Argumentation*. Dordrecht: Springer Netherlands. 519–560.

Ward, H. N.; Mills, B.; Brooks, D. J.; Troja, D.; and Khakhalin, A. S. 2021. AI solutions for drafting in Magic: the Gathering. In *2021 IEEE Conference on Games (CoG)*, 1–8. Institute of Electrical and Electronic Engineers.

Zhang, Y.; Fontaine, M. C.; Hoover, A. K.; and Nikolaidis, S. 2022. Deep surrogate assisted MAP-elites for automated hearthstone deckbuilding. In Fieldsend, J. E., ed., *GECCO '22: Proceedings of the Genetic and Evolutionary Computation Conference*, 158–167. Association for Computing Machinery.

Zhong, M.; Liu, G.; Li, H.; Kuang, J.; Zeng, J.; and Wang, M. 2022. CodeGen-Test: An automatic code generation model integrating program test information.

Zhong, R.; Yu, T.; and Klein, D. 2020. Semantic evaluation for text-to-SQL with distilled test suites. In Webber, B.; Cohn, T.; He, Y.; and Liu, Y., eds., *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 396–411. Association for Computational Linguistics.