Provable Limitations of Acquiring Meaning from Ungrounded Form: What Will Future Language Models Understand?

William Merrill* **Yoav Goldberg*†** **Roy Schwartz‡** **Noah A. Smith*§**

*Allen Institute for AI, United States †Bar Ilan University, Israel

‡Hebrew University of Jerusalem, Israel §University of Washington, United States

{willm,yoavg,roys,noah}@allenai.org

Abstract

Language models trained on billions of tokens have recently led to unprecedented results on many NLP tasks. This success raises the question of whether, in principle, a system can ever ''understand'' raw text without access to some form of grounding. We formally investigate the abilities of ungrounded systems to acquire meaning. Our analysis focuses on the role of ''assertions'': textual contexts that provide indirect clues about the underlying semantics. We study whether assertions enable a system to emulate representations preserving semantic relations like equivalence. We find that assertions enable semantic emulation of languages that satisfy a strong notion of semantic transparency. However, for classes of languages where the same expression can take different values in different contexts, we show that emulation can become uncomputable. Finally, we discuss differences between our formal model and natural language, exploring how our results generalize to a modal setting and other semantic relations. Together, our results suggest that assertions in code or language do not provide sufficient signal to fully emulate semantic representations. We formalize ways in which ungrounded language models appear to be fundamentally limited in their ability to ''understand''.

1 Introduction

Recently, language models trained on huge datasets of raw text have pushed the limits of natural language processing (Devlin et al., 2019; Raffel et al., 2019; Brown et al., 2020, among others). Such systems transcend the *expert system* paradigm, where rules about language and meaning are hardcoded into a system, as well as the *supervised learning* paradigm, where a notion of meaning is provided through ground-truth labels. Rather, analysis of massive language models has revealed that, to some degree, knowledge of syntactic and semantic dependencies can emerge *without explicit supervision* (Rogers et al., 2020; Tenney et al., 2019). This knowledge can then be transferred to a variety of downstream NLP tasks.

Yet, today's NLP systems built on large language models still fall short of human-level general understanding (Yogatama et al., 2019; Zhang et al., 2020). Brown et al. (2020) discuss the limitations of their GPT-3 language model compared with humans, suggesting that:

> Scaling up any LM-like model . . . may eventually run into (or could already be running into) the limits of the pretraining objective.

This possibility raises an interesting theoretical question. What are the fundamental limits of learning meaning from language modeling, even assuming a perfect learner with access to unlimited data? Recently, Bender and Koller (2020) argued that achieving true natural language understanding from text alone is impossible, and that, to really get at meaning, some type of semantic grounding is necessary.[1] Their style of argumentation largely focused on developing thought experiments, rather than making formal arguments.

One thought experiment featuring prominently in Bender and Koller (2020) was the task of learning to understand a programming language's semantics from raw code. Here, understanding was defined as fully emulating a compiler. This setup has clear parallels to learning to understand natural language, although the more well-defined nature of programming languages makes them easier to reason about. Bender and Koller (2020) argue that emulation is difficult in this setting, and perhaps impossible, because the source code alone contains no information about how it should be interpreted to create outputs. One counterpoint

[1] See Michael (2020) for a summary of the informal discussion around Bender and Koller (2020), much of which took place on social media.

Transactions of the Association for Computational Linguistics, vol. 9, pp. 1047–1060, 2021. https://doi.org/10.1162/tacl_a_00412
Action Editor: Mark-Jan Nederhof. Submission batch: 9/2020; Revision batch: 4/2021; Published 9/2021.

raised by the paper, as well as others (Michael 2020; Potts, 2020), is the existence of unit tests, with *assertions* encoding examples of input/output pairs for blocks of code.[2] For example, systematically observing blocks like `x = 3; assert x == 3` could let a system bootstrap the semantics of variable assignment, because a programmer is likely to write assertions that will pass. These assertions constitute a form of implicit grounding embedded within language modeling by the pragmatic concerns of programmers, and they could potentially be leveraged to emulate a compiler.[3] However, it is not immediately clear if unit tests provide ''enough'' supervision to do this, even with unlimited data.

Viewing the debate about the power of assertions as central to the larger philosophical question, we aim to clarify it in more formal terms. In this paper, we formally study whether observing a generalized notion of assertions can allow a system to ''understand'' strings. An assertion is a query about whether two strings evaluate to the same value within a fixed context. This is motivated by the role of assertions in unit tests, where asserting two expressions are equal suggests that they have the same value within the test.

While assertions are directly motivated by the compiler thought experiment, they also have analogs in natural language, where sentences make assertions about the world, and it is reasonable to expect some form of bias towards true statements (Potts, 2020). Indeed, this is one of Grice's Maxims (Grice, 1975): a set of basic principles proposed to govern the pragmatics of natural language. For example, the truth conditions of *This cat is the cat that Mary owns* verify that two cats in the world identified in distinct ways are the same entity. In general, we might expect a sentence to appear with higher frequency if its truth conditions hold within its context, similar to an assertion in code, although of course there will also be other factors governing sentence frequency besides this. In this sense, the example sentence resembles the Python statement `assert cat1 == cat2`, where `cat1` and `cat2` are two `Cat` objects. See Section 6 for more discussion of how assertions and other formal concepts translate to natural language. We will generalize assertions to an abstract formal language context, allowing us to study how they can be used to emulate semantic relations.

Our findings are as follows. If every expression in a language has the same value in every valid context, then the language can be emulated using a finite number of assertion queries (Section 4). However, we construct a class of languages where expressions can take different values in different contexts, and where assertions do not enable emulation, i.e., infinite queries would be required (Section 5). Intuitively, this means that assertions do not provide enough signal for a Turing-complete emulator to fully ''understand'' languages from this class. We go on to discuss differences between our formal model and the less well-defined context of natural language (Section 6).

These results provide a formal way to characterize upper bounds on whether it is possible to emulate the semantics of a language from distributional properties of strings. Within our framework, in certain settings, we find that meaning cannot be learned from text alone. We strengthen claims made by Bender and Koller (2020) that assertions in code do not necessarily provide sufficient signal for a language model to emulate understanding. We do not make strong claims about how these results transfer to natural language, although we expect that the added complexity of natural language would make it, if anything, more difficult to ''understand'' than code.[4]

2 Preliminaries

Let $L \subseteq \Sigma^\star$ denote a formal language over alphabet Σ. We will use λ to denote the empty string.

Let $(\Sigma^\star)^2$ denote the Cartesian product of $\Sigma^\star$ with itself; that is, the set of all pairs of strings. Resembling Clark (2010), we refer to a tuple $\langle l, r\rangle \in (\Sigma^\star)^2$ as a syntactic *context*. We also use other symbols to refer to a context (e.g., $\kappa = \langle l, r\rangle$). We denote by λ^2 the empty context $\langle \lambda, \lambda\rangle$.

2.1 Meaning

We will model formal languages not just as sets of strings, but as having an associated semantics.[5]

[2]Unit tests are blocks of code in a software project that are designed to test whether the core code is behaving correctly.

[3]Contexts like assertions can be seen as an argument in favor of the distributional hypothesis (Harris, 1954).

[4]Appendix C documents and motivates conceptual changes since the original arXiv version of the paper.

[5]We slightly abuse notation by using L to refer to both a set of strings, and a set of strings paired with a denotation function, which could be written more verbosely as $\langle L, [\![\cdot]\!]_L\rangle$.

Specifically, we assume the existence of a *denotational semantics* over every substring of L, which we now elaborate on. Let Y be a countable set of referents. First, we will say that some $e \in \Sigma^\star$ is a valid *expression* within the context $\kappa = \langle l, r \rangle$ if there exists some contextual denotation $[\![e \mid \kappa]\!]_L \in Y$. Intuitively, this represents the value of e when it occurs in the larger context $ler \in L$. We will also use the notation $[\![e \mid l, r]\!]_L$ where convenient. We will reserve $\emptyset \in Y$ as a special null symbol, defining $[\![e \mid \kappa]\!]_L = \emptyset$ iff e is not a valid expression in the context κ.[6]

Each context $\kappa \in (\Sigma^\star)^2$ also has a *support*, or set of expressions that are valid within it:

$$\text{supp}_L(\kappa) = \{e \in \Sigma^\star \mid [\![e \mid \kappa]\!]_L \neq \emptyset\}.$$

Example Let L be a language of integers along with the `+` operator, for example, `2 + 2`. Y is simply the integers. We take $[\![e \mid \kappa]\!]_L$ to map e to its standard arithmetic interpretation, namely, $[\![\texttt{2 + 6} \mid \lambda, \texttt{+ 4}]\!]_L = 8$. We take expressions that are not conventionally well-formed to be invalid: for example, $[\![\texttt{+} \mid \lambda, \texttt{+}]\!]_L = \emptyset$. Finally, let $\kappa = \langle \lambda, \texttt{+ 4} \rangle$. Then $\text{supp}_L(\kappa) = L$, since any valid expression can occur within κ.

2.2 Strong Transparency

As defined above, we make very few assumptions about denotations. They are not necessarily compositional, and expressions may take different referents in different contexts. However, we saw in the integer expression language that the meanings of an expression did not depend on its context. We now define a property formalizing this idea.

Definition 1 (Strong transparency) L is *strongly transparent* iff, for all $e \in \Sigma^\star$, $\kappa \in (\Sigma^\star)^2$, either $[\![e \mid \kappa]\!]_L = [\![e \mid \lambda^2]\!]_L \neq \emptyset$, or $[\![e \mid \kappa]\!]_L = \emptyset$.

Informally, strong transparency says each e has a well-defined denotation that exists independent of context, and that this simple denotation can be ''plugged into'' any context. Our previous example expression `2 + 6` is strongly transparent because it can be said to have a well-defined value 8 independent of its context. We could break strong transparency by adding bound variables to the language, for example, `x = 2; x + 6` in Python. In this case, $[\![\texttt{x} \mid \kappa]\!]_L$ non-vacuously depends on κ.

Strong transparency resembles referential transparency (Whitehead and Russell, 1925–1927), but is a stronger condition, in that it does not allow the same name to *ever* refer to different values. For example, for a Python program, strong transparency does not allow assigning local variables within a function, even if the function output would remain completely specified by its inputs.

2.3 Assertion Queries

We now define an oracle function providing assertion information about expressions in L, resembling `assert e1 == e2` for two Python expressions `e1`, `e2`. A system is granted access to this function, and it can make *assertion queries* to it in order to learn about the semantics of L.[7] An assertion query tells us whether two expressions e, e' are equivalent within the context κ.

Definition 2 (Assertion oracle) For $e, e', \in \Sigma^\star$ and $\kappa \in (\Sigma^\star)^2$, define the *assertion oracle*

$$\aleph_L(e, e' \mid \kappa) = \begin{cases} 1 & \text{if } [\![e \mid \kappa]\!]_L = [\![e' \mid \kappa]\!]_L \\ 0 & \text{otherwise.} \end{cases}$$

Recall that we defined $[\![e \mid \kappa]\!]_L = \emptyset$ if e is not valid in the context κ. In our example language of integer expressions, for all κ, $\aleph_L(\texttt{4}, \texttt{2 + 2} \mid \kappa) = 1$, since $4 = 2 + 2$. The computational power of this oracle depends on the complexity of the underlying semantics: For arbitrary semantics, it can become uncomputable. In this paper, though, we focus on classes of languages for which the denotation function and assertion oracle are computable.

The $\aleph_L$ oracle is motivated by assertion statements in programming languages, which occur naturally in environments like unit tests. The distribution of strings in a corpus of code should capture some notion of this oracle, since a programmer is more likely to assert two expressions are equal if

[6] Our simple model of denotations does not reflect the full range of semantic theories that have been proposed for natural language. In particular, our denotations $[\![e \mid \kappa]\!]_L$ depend only on the linguistic context κ rather than any external world state. This differs substantially from how truth conditions are traditionally conceptualized in formal semantics (Heim and Kratzer, 1998). For example, in our framework, the referent of English $[\![\textit{the dog} \mid \kappa]\!]_L$ must be fixed with no regard for the extralinguistic context. Section 6 further contrasts our setup with the richer semantics of natural language.

[7] This resembles the role of queries in classical grammar induction works (e.g., Angluin, 1987).

they are expected to have the same value. Our goal is to study the limits of understanding achievable from raw text, so we consider an ‘‘upper bound’’ setup by assuming a system has full access to $\aleph_L$. Can the system use this powerful oracle to emulate the underlying semantics?

2.4 Turing Machines

Our notion of language understanding will be based around the idea of emulation, which in turn requires a model of computational realizability. We will use Turing machines (Turing, 1936) as a model of universal computation. We write $\mu(e)$ for the output of Turing machine μ evaluated on input $e \in \Sigma^\star$. We will also define an oracle Turing machine as a standard Turing machine that can compute a blackbox ‘‘oracle’’ function f as a subroutine. We imagine the machine has a special *query* instruction and tape. After writing x to the query tape and executing the query instruction, the query tape will contain $f(x)$. We will write $\mu_f(e)$ for the Turing machine μ evaluated on input e with oracle access to f. In the case where $f = \aleph_L$, we will simply write $\mu_L(e)$. Whereas, in computability theory, oracle Turing machines are generally leveraged to make reductions from uncomputable problems, here we will use them to formalize the ability of an emulator to make assertion queries about L. This oracle provides additional power because these queries contain additional information beyond that encoded in the input expression.

3 Research Question: Do Assertions Enable Emulation?

There is a long history in AI of trying to define and measure understanding. Turing (1950) constitutes an early behaviorist perspective; more recent approaches tend to emphasize not just an external view of a system’s behavior, but also ‘‘how it is achieved’’ (Levesque, 2014). Understanding can be behaviorally diagnosed in neural models by evaluating them on benchmarks (Wang et al., 2018). An alternate approach is probing (Adi et al., 2017; Conneau et al., 2018; Hupkes and Zuidema, 2018; Hewitt and Liang, 2019; Belinkov and Glass 2019), which investigates *how directly* a model’s representations encode semantic relations by measuring if they can be easily decoded from

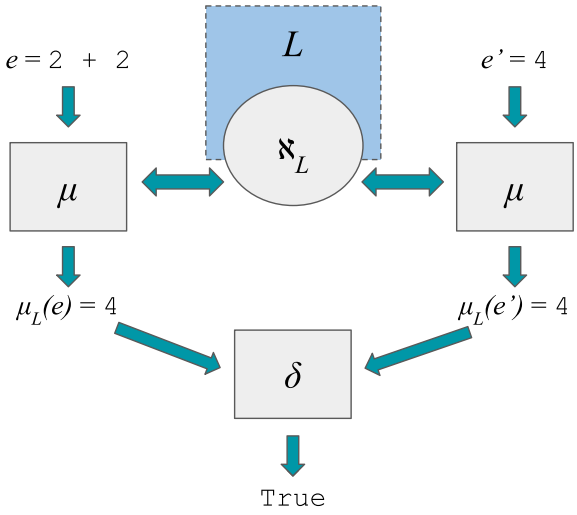

Figure 1: An illustration of Definition 3. μ emulates a representation of each expression using assertion queries. Then, δ compares the emulated representations to determine equivalence.

them. Similarly, we take the position that systems are capable of understanding if they *emulate* representations that are isomorphic to underlying meaning under important semantic relations like equivalence. We will formalize this in Question 1, which asks whether such emulation is possible using assertions.

Definition 3 ($\aleph$-emulation) A class of languages $\mathcal{L}$ over Σ is $\aleph$-*emulatable* if there exists an oracle Turing machine μ and standard Turing machine δ such that, for all $L \in \mathcal{L}$, $\kappa \in (\Sigma^\star)^2$, and $e, e' \in \text{supp}_L(\kappa)$,

$$[\![e \mid \kappa]\!]_L = [\![e' \mid \kappa]\!]_L \iff \delta\big(\mu_L(e), \mu_L(e') \mid \kappa\big).$$

μ can be thought of as an emulator that evaluates expressions, whereas δ receives two values and decides whether they are equal. Crucially, only μ has direct access to $\aleph_L$. δ can only use information from the oracle to the extent that it is encoded in the representations $\mu_L(e)$ and $\mu_L(e')$.

Definition 3 formulates emulation as a decision problem, as is typical in theoretical computer science. Equivalently, δ can be replaced by a computable function ρ such that $\rho(\mu_L(e) \mid \kappa)$ *evaluates* $\mu_L(e)$ in context κ, that is, its output string is isomorphic to $[\![e \mid \kappa]\!]_L$ under $=$. The functions δ and ρ are Turing-reducible to each other, implying that if one definition is satisfied, so is the other.

```python
from typing import Callable

AssertType = Callable[[str, str, str, str], bool]

def emulate(expr: str, asserteq: AssertType) -> int:
    for idx, cand in enumerate(all_strings()):
            if asserteq(expr, cand, "", ""):
                return idx
```

Figure 2: `emulate` implements an emulator μ. Let `all_strings` be an iterable enumerating all strings in $\Sigma^\star$. We provide a concrete implementation of `all_strings` in Figure 5.

With our definition of emulation in place, we can formally state the research question:

Question 1 *For a class of languages $\mathcal{L}$, is $\mathcal{L}$ $\aleph$-emulatable?*

How does Question 1 relate to understanding in large language models? We imagine that, with sufficiently large amounts of data, the frequencies of strings in L carry enough signal such that the language model objective ''supervises'' access to $\aleph_L$. Thus, $\mu_L(e)$ can be thought of as the language model representation of an expression e. We then hope to recover underlying semantic relations from the representations produced by the language model via some function δ. The class $\mathcal{L}$ corresponds to a set of hypothesis languages over which the language model must search for the true L. We will see that whether emulation is possible will depend on the properties of $\mathcal{L}$.

Stepping back, Question 1 bears on the role of assertions raised by Bender and Koller (2020). Does observing assertions allow a Turing-complete system to emulate a compiler? In more general terms, are assertions powerful enough implicit grounding to achieve representations that encode the denotational semantics of a language?

4 Strong Transparency

We first consider the case where the language being learned is known to be strongly transparent. Let TRANSPARENT denote the class of strongly transparent languages. We will show that TRANSPARENT is $\aleph$-emulatable. The core idea of the proof is to construct a canonical form for each expression. The canonical form is the first expression in a lexicographic ordering that the assertion oracle deems equivalent to the target expression. For technical reasons, the emulator returns the index of this string under the lexicographic order.

Theorem 1 TRANSPARENT *is $\aleph$-emulatable.*

Proof. As Python is Turing-complete, we write $\mu : \Sigma^\star \to \mathbb{N}$ as a Python function `emulate` in Figure 2. The function receives as input an expression `expr` and a callback function `asserteq` to an oracle computing $\aleph_L$. For each $e \in \Sigma^\star$, there exists $e^\star \in \Sigma^\star$ such that $\aleph_L(e, e^\star \mid \lambda^2) = 1$. In the ''worst case'', this holds when $e^\star = e$ by symmetry. By construction, `all_strings` reaches all strings in finite time. Therefore, the number of loop iterations before reaching $e^\star$ is finite. We can conclude that `emulate` halts on every $e \in \Sigma^\star$, establishing that it is computable.

Now, we move towards justifying that the emulation is correct for every $\kappa \in (\Sigma^\star)^2$. We note that δ is simply the indicator function for equality over the natural numbers:

$$\delta(m, m' \mid \kappa) = \begin{cases} 1 & \text{if } m = m' \\ 0 & \text{otherwise.} \end{cases}$$

The function `emulate` outputs $i \in \mathbb{N}$, the index of the first string $e^\star$ such that $[\![e \mid \lambda^2]\!]_L = [\![e^\star \mid \lambda^2]\!]_L$. Now, let $e, e' \in \text{supp}_L(\kappa)$ be different inputs to μ. Because the enumeration order of the `for` loop is fixed across computation of $\mu_L(e)$ and $\mu_L(e')$:

$$\begin{aligned} \mu_L(e) = \mu_L(e') &\iff [\![e \mid \lambda^2]\!]_L = [\![e^\star \mid \lambda^2]\!]_L \\ &\quad \wedge [\![e' \mid \lambda^2]\!]_L = [\![e^\star \mid \lambda^2]\!]_L \\ &\iff [\![e \mid \lambda^2]\!]_L = [\![e' \mid \lambda^2]\!]_L \\ &\iff [\![e \mid \kappa]\!]_L = [\![e' \mid \kappa]\!]_L, \end{aligned}$$

where the last step follows by strong transparency. We conclude that the conditions for emulation (Definition 3) are fully satisfied. □

Through a simple construction, we have shown it is possible to emulate meaning from assertion queries for languages with strongly transparent

```
def leq() -> bool:
    return n < M
print(leq())
```

```
def leq() -> bool:
    return n < M
print(True)
```

Figure 3: Templates for strings in L_m, for $m \in \mathbb{N} \cup \{\infty\}$. `M` evaluates to m in all strings, while other expressions are evaluated according to Python 3.8 semantics. The metavariable n ranges over $\mathbb{N}$ to form different strings in L_m, and is serialized as a decimal string.

semantics. The number of bits in the emulated representation $\mu_L(e)$ is linear in the size of e. In the next section, we consider what happens without strong transparency, where, among other complexities, values can be bound to variables, complicating the construction used in Theorem 1.

5 General Case

Requiring strong transparency precludes a broad class of linguistic patterns allowing an expression to refer to different values in different contexts. For example, this includes assigning variable or function names in Python, or binding pronouns in natural language. These constructions can make emulation impossible to achieve from assertions. We will construct a class of languages based on Python where emulation is uncomputable.

Definition 4 Let $\textsc{Leq} = \{L_m \mid m \in \mathbb{N} \cup \{\infty\}\}$, where strings in L_m are defined according to Figure 3. For semantics, we first define $[\![\texttt{M} \mid \kappa]\!]_{L_m} = m$. For any other $ler \in L_m$ that is a well-formed Python 3.8 expression, we define $[\![e \mid l, r]\!]_{L_m}$ as the value of e assigned by the Python interpreter in the context $\langle l, r\rangle$. For strings that are not valid Python expressions, define $[\![e \mid l, r]\!]_{L_m} = \emptyset$.

What does it take to emulate the expressions `leq()` and `True` in L_m? If we knew m, then we could emulate them by simply comparing $n < m$. However, it turns out that recovering m for any $L_m \in \textsc{Leq}$ is not possible with a fixed number of assertion queries. Formalizing this, we will show that Leq is not $\aleph$-emulatable.[8]

Theorem 2 Leq *is not* $\aleph$*-emulatable.*

Proof. Without loss of generality, we focus on the contexts for `leq()`[9] and `True` within `print`$(\cdot)$, each of which is parameterized by some value of n. Notationally, we identify each L_m with m, and each context with its parameter n. This enables shorthand like $[\![e \mid n]\!]_m$ for the denotation of the expression e in the context parameterized by n in L_m.

When $m = \infty$, it holds for all n that $\aleph_\infty(\texttt{leq()}, \texttt{True} \mid n) = 1$. To satisfy emulation of $e \in \{\texttt{leq()}, \texttt{True}\}$, μ_∞ makes a finite number of assertion queries

$$\aleph_\infty(\texttt{leq()}, \texttt{True} \mid n_i).$$

for some sequence of contexts $n_1, \cdots, n_q$, which we assume without loss of generality is sorted in increasing order. We can adversarially construct $m' \neq \infty$ such that all these queries are the same, and thus $\mu_\infty(e) = \mu_{m'}(e)$ for both e. To implement this, we simply set $m' = n_q + 1$. Since $\mu_\infty(e) = \mu_{m'}(e)$, we conclude that, for all n,

$$\delta(\mu_{m'}(\texttt{leq()}), \mu_{m'}(\texttt{True}) \mid n) = \delta(\mu_\infty(\texttt{leq()}), \mu_\infty(\texttt{True}) \mid n).$$

On the other hand, consider $n > n_q$. In this case,

$$[\![\texttt{leq()} \mid n]\!]_{m'} = \texttt{False}$$
$$[\![\texttt{leq()} \mid n]\!]_\infty = \texttt{True},$$

which can be rewritten as

$$[\![\texttt{leq()} \mid n]\!]_{m'} \neq [\![\texttt{True} \mid n]\!]_{m'}$$
$$[\![\texttt{leq()} \mid n]\!]_\infty = [\![\texttt{True} \mid n]\!]_\infty.$$

Therefore, the conditions of $\aleph$-emulation (Definition 3) cannot be satisfied for both $L_{m'}$ and L_∞. This implies that Leq is not $\aleph$-emulatable. □

5.1 Discussion

We briefly summarize this result in less formal terms. Leq contains languages L_m defined by Figure 3. Every program in each L_m is easily computable. With knowledge of the Python interpreter and m, any agent could execute all of

[8] Another example of a non-$\aleph$-emulatable language takes `M` to be a finite list of integers and replaces $n <$ `M` with n `in` `M`.

[9] The only "valid" context for `leq()` is within `print`$(\cdot)$. The denotation of `leq()` when it occurs next to `def` is $\emptyset$.

There is a number.	There is a number.
n is less than it.	Zero equals one.

Figure 4: An informal construction adapting the program templates in Figure 3 to English. Under our framework, two sentences are considered equivalent if they are true in exactly the same set of contexts. If the number is allowed to be ∞, this cannot be done in general for the final lines of each template.

these programs. This can be formalized by observing that, for a fixed m, the class $\{L_m\}$ is $\aleph$-emulatable. Rather, what we have shown is that, with finite time, it is impossible for an ungrounded agent to emulate L_m *using assertion queries* when m is unknown in advance. In other words, without prior knowledge of m, no algorithm can use assertions to disambiguate which notion of $=$ is used by L_m from the infinite other possibilities. In a rough sense, m can be thought of as a cryptographic key enabling linguistic understanding: agents that know m can directly emulate L_m, but agents without it cannot, at least using assertions.[10]

Theorem 2 does not use the fact that δ must be computable, as opposed to an arbitrary function. Even if δ is an arbitrary function, it could not disambiguate whether m halts based on queries.

It is more precise to state Theorem 2 in a formal language, but an argument similar to Theorem 2 can be adapted to a natural language like English. An example is shown in Figure 4, where we define the meaning of a sentence as its truth conditions, and we imagine the class of candidate languages is formed by varying the unspecified *number*, which can potentially be ∞. Deciding if n is less than it has the same truth conditions as *Zero equals one* is equivalent to comparing `leq()` and `True`. A system must necessarily fail to emulate the semantics of these expressions in some context, for some secret number. The rest of the paper further explores the implications and limitations of applying our formal model to natural language.

[10] Alternatively, we can take a more complexity-theoretic perspective by measuring the number of queries needed to emulate up to a bounded context size. Fix a maximum n. Then we can use binary search with $\mathcal{O}(\log n)$ queries to find the value of m. Since the number of context bits is $\mathcal{O}(\log n)$, the numbers of queries is $\mathcal{O}(|\kappa|)$, beating the $\mathcal{O}(|\Sigma|^{|\kappa|})$ query complexity achievable by brute force. This perspective somewhat resembles Pratt-Hartmann and Third (2006) and other work in semantic complexity theory on the computational complexity of evaluating fragments of natural language.

6 Towards Natural Language

As discussed in Section 1, our results are inspired by the thought experiment of whether a language model can use raw code to learn a compiler. A goal of this, of course, is to examine whether understanding can be acquired from natural language text in a simplified setting. In principle, our formal results can bear on this broader question about natural language, although some differences emerge when extending the results to a less well-defined setting. In many cases, these differences appear to make the task of learning meaning harder, suggesting that our negative claim in a simpler setting (Theorem 2) may still hold as an impossibility result. We now discuss some points of difference between our formal model and natural language.

Truth Conditions There are connections between our framework and the concepts of truth values and truth conditions in linguistic semantics. For a Boolean-valued expression e, a truth value corresponds to computing $[\![e \mid \kappa]\!]_L$ in a fixed context. On the other hand, truth conditions correspond roughly to a function computing $[\![e \mid \kappa]\!]_L$ for any κ. A crucial difference, though, is that these conditions cannot be *intensional* (Von Fintel and Heim, 2011), that is, they are not functions of the world state, but rather of the linguistic context only. In this sense, emulation corresponds to recovering the ability to resolve non-intensional truth conditions of sentences. This model is natural for formalizing a closed programming language environment, for example, with no environment variables or user input, since in this case the program state is specified completely by the linguistic context. On the other hand, English has common elements like *that* whose meaning can change depending on world state external to language. Perhaps allowing such elements would only make understanding more difficult; or, arguably, generally impossible, since there is no way for the model to observe the grounding world state using only an assertion oracle. We are inclined to

believe that, since such changes would make understanding more difficult, Theorem 2 would still hold as an impossibility result. However, future work would be needed to make this idea precise.

Possible Worlds In the last paragraph, we discussed how mutable world state is an additional complexity of natural language compared to our setup. Similarly, speakers of natural languages have imperfect information about the world around them, which can be captured by modeling the referent of an expression over a set of *possible* worlds, rather than within a specific evaluation context. In Appendix A, we explore to what degree this setting makes the task of learning to understand more difficult. In adapting our model to this context, the assertion oracle must become ''modal'' in the sense that it quantifies over sets of worlds. We explore two different models of modality for the oracle, corresponding to different physical interpretations. In one case, Theorem 1 and Theorem 2 apply analogously, while, in the other, emulation becomes an ill-defined problem.

Denotation vs. Intent Bender and Koller (2020) distinguish between *standing meaning* and *communicative intent*, reflecting a distinction between denotational semantics and other pragmatic intentions that a speaker has in producing an utterance. In this paper, it is most straightforward to take $[\![e \mid \kappa]\!]_L$ to reflect standing meaning. In principle, we could imagine that it represents the speaker's communicative intent, and that an omniscient oracle $\aleph_L$ can reveal information about the speaker's intents to the system. Even with this unrealistically powerful oracle, Theorem 2 says that the system cannot emulate the speaker's intents.

Competence vs. Performance Chomsky (1965) differentiates competence and performance in linguistic theory, where competence corresponds roughly to the correct algorithmic modeling of a linguistic process, and performance describes its implementation subject to resource constraints like memory. Arguably, agents might be said to understand language if they are competent in this sense, even if they sometimes make performance errors. In contrast, our definition of emulation (Definition 3) permits no performance errors. In future work, it would be interesting to adapt an approximate notion of emulation that tolerates performance errors in order to more closely target understanding in a sense reflecting competence.

Other Relations Theorem 1 and Theorem 2 investigate whether $\aleph_L$ can be used to emulate meaning representations that preserve an equivalence relation. While equivalence is an important part of semantics, other semantic relations like entailment are also necessary for language understanding. In Appendix B, we show a generalization of Theorem 5 extends to *any* semantic relation. In other words, referential transparency also enables emulation of relations besides $=$.

Other Oracles We believe assertions are a fairly general model of the types of semantics encoded in unsupervised learning resulting from a pragmatic bias for truth; however, it is possible other information is also represented, resulting from other pragmatic biases governing language usage and dataset creation. This additional information could be formalized as access to additional oracles. It would be exciting to formalize the power of multimodal setups by analyzing the interactions of oracles enabled by different input modalities.

7 Stepping Back

In this work, we formalized an argument that was raised by Bender and Koller (2020) as a thought experiment. Bender and Koller (2020) question whether unsupervised training objectives are the right goal to target for achieving natural language understanding. If meaning is defined as identifying which object in the real world, or which set of situations, a linguistic element refers to, then, in a direct sense, an ungrounded system cannot understand meaning. But Bender and Koller (2020) go farther than this, claiming that an ungrounded system cannot even *emulate* understanding because it is not clear how a system should learn to interpret strings, even if it can model their distribution. We formalize this idea of emulation as $\aleph$-emulation.

One counterargument mentioned by Bender and Koller (2020) is that indirect forms of grounding do exist in programming and natural language, which we formalize as assertions. The syntactic distributions of statements like `assert` allow us to indirectly observe semantic relations over the denotations. Assertions are one way that the distribution of strings in a corpus is not blind to their semantics. By studying them, we study whether this indirect grounding enables a computational system to emulate the underlying semantic relations.

Key Takeaways While assertions allow a system to emulate semantic relations in simple cases where the semantics are referentially transparent, we find that linguistic constructs like variable binding bring this task in conflict with the fundamental laws of computability. In other words, under our formal model of meaning and emulation, it is not just intractable for an ungrounded system to emulate understanding of a formal language, but, in some cases, *impossible*. We provide constructive examples where understanding must necessarily break down. We present these results in a well-defined framework building off formal approaches in logic, linguistics, and computer science. While we do not prove anything about natural languages, we do show that ungrounded models must fail to emulate equivalence in a very simple setting. A similar result likely extends to natural language understanding as well, which among other things, requires modeling referential identity (e.g., for sentences like *Manny is the cat*). Further, we believe much of our framework can be readily adopted in other works formalizing understanding in Turing-complete systems.

Open Questions In this work, we have focused on utterances, by default, as opposed to *dialogues*. An exciting extension would be to formalize a dialogue between two speakers, interrupted by the ''octopus'' of Bender and Koller (2020).[11] Existing theories of discourse could potentially be synthesized with this framework. What linguistic properties besides referential transparency relate to emulatability? Can this framework be extended to formalize multimodal setups, where multiple oracles from different domains can potentially be combined to gain additional power? Finally, is there a natural way to relax our standard of emulation towards a probabilistic definition, and how would this change the results?

Acknowledgments

We thank Mark-Jan Nederhof for his excellent suggestions. We also thank Dana Angluin, Matt Gardner, Eran Yahav, Zachary Tatlock, Kyle Richardson, Ruiqi Zhong, Samuel Bowman, Christopher Potts, Thomas Icard, and Zhaofeng Wu for their feedback on various versions of this work. Further thanks to our anonymous reviewers and researchers at the Allen Institute for AI and UW NLP. Finally, we appreciate the lively online discussion of the paper, which informed updates to the camera-ready version.

[11]The octopus thought experiment imagines a deep-sea octopus O observes a dialogue between two humans by intercepting an underwater cable. Could O learn to emulate the role of one of the speakers without exposure to life on land?

References

Yossi Adi, Einat Kermany, Yonatan Belinkov, Ofer Lavi, and Yoav Goldberg. 2017. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. In *5th International Conference on Learning Representations, ICLR* 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net.

Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Yonatan Belinkov and James Glass. 2019. Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72. https://doi.org/10.1162/tacl_a_00254

Emily M. Bender and Alexander Koller. 2020. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.463

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. The arXiv is: 2005.14165.

Noam Chomsky. 1965. *Aspects of the Theory of Syntax*, volume 11. MIT Press.

Alexander Clark. 2010. Three learnable models for the description of language. In *Language and Automata Theory and Applications*, pages 16–31, Berlin, Heidelberg. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-13089-2_2

Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia. Association for Computational Linguistics. https://doi.org/10.18653/v1/P18-1198

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Herbert P Grice. 1975. Logic and conversation. In *Speech Acts*, pages 41–58. Brill. https://doi.org/10.1163/9789004368811_003

Zellig S. Harris. 1954. Distributional structure. *WORD*, 10(2–3):146–162.

Irene Heim and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Blackwell.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743, Hong Kong, China. Association for Computational Linguistics. https://doi.org/10.18653/v1/D19-1275

Laurence R. Horn and Heinrich Wansing. 2020. Negation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*, spring 2020 edition. Metaphysics Research Lab, Stanford University.

Dieuwke Hupkes and Willem Zuidema. 2018. Visualisation and 'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure (extended abstract). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5617–5621. International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2018/796

Hector Levesque. 2014. On our best behaviour. *Artificial Intelligence*, 212. https://doi.org/10.1016/j.artint.2014.03.007

Julian Michael. 2020. To dissect an octopus: Making sense of the form/meaning debate.

Christopher Potts. 2020. Is it possible for language models to achieve understanding? https://doi.org/10.1305/ndjfl/1153858644

Ian Pratt-Hartmann and Allan Third. 2006. More fragments of language. *Notre Dame Journal of Formal Logic*, 47(2):151–177.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. The arXiv: 1910.10683.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A primer in BERTology: What we know about how BERT works. arXiv: 2002.12327. https://doi.org/10.1162/tacl_a_00349

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601, Florence, Italy. Association for Computational Linguistics. https://doi.org/10.18653/v1/P19-1452

Alan M. Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58(345–363):5.

Alan M. Turing. 1950. Computing machinery and intelligence. *Mind*, LIX(236):433–460. https://doi.org/10.1093/mind/LIX.236.433

Kai Von Fintel and Irene Heim. 2011. Intensional semantics. *Unpublished Lecture Notes*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium. Association for Computational Linguistics. https://doi.org/10.18653/v1/W18-5446

Alfred North Whitehead and Bertrand Russell. 1925–1927. *Principia Mathematica*, Cambridge University Press.

Yoad Winter. 2016. *Elements of Formal Semantics: An Introduction to the Mathematical Theory of Meaning in Natural Language*. Edinburgh University Press.

Dani Yogatama, Cyprien de Masson d'Autume, Jerome Connor, Tomas Kocisky, Mike Chrzanowski, Lingpeng Kong, Angeliki Lazaridou, Wang Ling, Lei Yu, Chris Dyer, and Phil Blunsom. 2019. Learning and evaluating general linguistic intelligence. arXiv: 1901.11373.

Hongming Zhang, Xinran Zhao, and Yangqiu Song. 2020. WinoWhy: A deep diagnosis of essential commonsense knowledge for answering Winograd schema challenge. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5736–5745, Online. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.508

A Multiple Worlds

Programs execute in well-defined environments with a clear state. Speakers of natural language, on the other hand, have imperfect information and beliefs about the world around them. Thus, it can be more natural to model grounding context for language as a set of *possible worlds*, rather than a single world state. We formalize this in two different ways (with two different physical interpretations) and explore how it affects our results.

Let W be a set of all possible worlds. We redefine denotations to be *intensionalized* (Von Fintel and Heim, 2011), that is, we write $[\![e \mid \kappa]\!]^w$ as the denotation of e in the context κ, evaluated in world $w \in W$. Assume for simplicity that $Y = \{0, 1, \emptyset\}$. We will now introduce modal denotations and assertions using a generic *modal quantifier* $\odot$, which reduces a sequence of worlds to a boolean value according to some intensional predicate. This quantifier controls how multiple possible worlds are collapsed to form denotations and query outputs.

Definition 5 (Modal denotation) Let $\odot$ be a modal quantifier. For all $e \in \Sigma^\star$, $\kappa \in (\Sigma^\star)^2$, define

$$\odot[\![e \mid \kappa]\!]_L = \bigodot_{w \in W} [\![e \mid \kappa]\!]_L^w.$$

We will write the previously defined assertion oracle to apply in a specific world w, namely, $\aleph_L^w$. We also extend it to quantify over multiple worlds:

Definition 6 (Modal assertion) Let $\odot$ be a modal quantifier. For all $e \in \Sigma^\star$, $\kappa \in (\Sigma^\star)^2$, define

$$\odot\aleph_L(e, e' \mid \kappa) = \bigodot_{w \in W} \aleph_L^w(e, e' \mid \kappa).$$

Specifically, we consider $\odot = \{\Box, \Diamond\}$, corresponding to universal and existential quantifiers over worlds. Thus, $\Box$ can be thought of as as $\forall$ over worlds, and $\Diamond$ can be thought of as $\exists$. For either quantifier, if any $[\![e \mid \kappa]\!]_L^w = \emptyset$, we define $\odot[\![e \mid \kappa]\!]_L = \emptyset$ as well. Each quantifier will have a different physical interpretation. With universal quantification, we will find that results analogous to Theorem 1 and Theorem 2 hold. With existential quantification, it turns out that the equivalence class of μ is underspecified. In other words, not only is it impossible to compute an emulator with a finite number of assertion queries, but, even with infinite assertions, there is no consistent way to emulate the underlying modal semantics.

A.1 Universal Quantification

In the first case we let $\odot = \Box$. Two expressions are viewed as having the same meaning if they

are equivalent in every possible belief world. This is interpretable as observing text $L_\Box$ written by a single author whose belief state is represented by multiple possible worlds. The author only asserts a statement is true if it is consistent across all worlds that they believe are possible.

In this setting, we will show that the modal assertion oracle uniquely specifies a modal denotation for each expression, up to isomorphism. In other words, as with the non-modal assertion oracle, each assertion query would let us decide some relation between two expressions. Thus, the same results for the non-modal setting discussed in the main body of the paper will also hold here.

Theorem 3 *Consider* $e, e' \in \Sigma^\star$ *and any context* $\kappa \in (\Sigma^\star)^2$ *such that* $\Box[\![e \mid \kappa]\!]_L \neq \emptyset$ *and* $\Box[\![e' \mid \kappa]\!]_L \neq \emptyset$*. Then,*

$$\Box[\![e \mid \kappa]\!]_L = \Box[\![e' \mid \kappa]\!]_L \iff \Box\aleph_L(e, e' \mid \kappa).$$

Proof.

$$\begin{aligned}
&\Box[\![e \mid \kappa]\!]_L = \Box[\![e' \mid \kappa]\!]_L \\
&\quad\iff \bigwedge_{w \in W} [\![e \mid \kappa]\!]_L^w = \bigwedge_{w \in W} [\![e' \mid \kappa]\!]_L^w \\
&\quad\iff \bigwedge_{w \in W} \left([\![e \mid \kappa]\!]_L^w = [\![e' \mid \kappa]\!]_L^w\right) \\
&\quad\iff \bigwedge_{w \in W} \aleph_L^w(e, e' \mid \kappa) \\
&\quad\iff \Box\aleph_L(e, e' \mid \kappa).
\end{aligned}$$

□

Crucial to this simple proof is the fact that $\wedge$ is distributive over $=$. This is specific to the quantifier being $\Box$. Theorem 3 implies that $\Box[\![e \mid \kappa]\!]_L$ can be recovered from modal assertion queries analogously to the non-modal case. Thus, results analogous to Theorem 1 and Theorem 2 apply for emulating $\Box[\![e \mid \kappa]\!]_L$ using queries to $\Box\aleph_L$.

A.2 Existential Quantification

In the second case we let $\odot = \Diamond$. Two expressions are viewed as having the same meaning if they are equivalent in *some* world. This is interpretable as observing a large dataset of text $L_\Diamond$ generated by many authors, each with a different single belief world w. In the corpus, we imagine two expressions can be asserted to be equivalent in some context if *any* of the authors would consider them to be equal in that context.

	e_1	e_2	$\aleph$	e_1	e_2	$\aleph$
w_1	0	0	1	0	0	1
w_2	0	0	1	0	1	0
$\Diamond$	0	0	1	0	1	1

Table 1: Two tables (separated by a thick line) representing two different versions of W. Within each table, each cell i, j in the main 2-by-2 grid contains the boolean value $[\![e_j \mid \kappa]\!]_L^{w_i}$. The column to the right contains $\aleph_L^{w_i}(e_1, e_2 \mid \kappa)$. The bottom row aggregates each column by quantifying $\Diamond$.

In this case, assertions do not even fully specify equivalence between the modal denotations. This is a stronger sense in which meaning cannot be emulated from assertion queries. Emulation is not just impossible with finite assertions, but mathematically underspecified.

Theorem 4 *There exist* $e, e' \in E(L)$ *and* $\kappa \in (\Sigma^\star)^2$ *such that* $\Diamond[\![e \mid \kappa]\!]_L \neq \emptyset$ *and* $\Diamond[\![e' \mid \kappa]\!]_L \neq \emptyset$*, and also* $\Diamond\aleph_L(e, e' \mid \kappa) = 1$ *is consistent with either* $\Diamond[\![e \mid \kappa]\!]_L = \Diamond[\![e' \mid \kappa]\!]_L$ *or* $\Diamond[\![e \mid \kappa]\!]_L \neq \Diamond[\![e' \mid \kappa]\!]_L$.

Proof. We construct an example with expressions e_1, e_2 in a single context κ. Fix $W = \{w_1, w_2\}$. Table 1 shows two versions of this modal setup. In both versions of the universe, $\Diamond\aleph_L(e, e' \mid \kappa) = 1$. However, on the left, $\Diamond[\![e \mid \kappa]\!]_L = \Diamond[\![e' \mid \kappa]\!]_L$, while, on the right, the opposite holds. So, with $\Diamond$, modal assertions do not uniquely determine equivalence of modal denotations. □

As an equivalence class for μ is not even well-defined by $\Diamond\aleph_L$, we cannot hope to compute it from queries. This is an even stronger sense in which emulation is impossible using assertions. On some level, this may be a natural model for language modeling corpora, which aggregate text from potentially inconsistent sources.

In summary, if assertions uniquely determine equivalence between denotations in a strongly transparent language, then we can expect to emulate representations preserving equivalence using assertions. Otherwise, there are various levels of formal challenges to emulating equivalence.

B Other Semantic Relations

Sections 4, 5, and A investigate whether $\aleph_L$ can be used to emulate meaning representations that

```python
from itertools import count, product
from typing import Iterable

def all_strings() -> Iterable[str]:
    for length in count():
        iterable = product(*[SIGMA for _ in range(length)])
        yield from ("".join(x) for x in iterable)
```

Figure 5: A concrete implementation of `all_strings`, which is referenced in Figure 2 and Figure 6.

```python
from typing import Callable, Dict, Tuple

AssertType = Callable[[str, str, str, str], bool]

def emulate(expr: str, assertrel: AssertType) -> Dict[Tuple[str, str], bool]:
    repres = {}
    for cand in all_strings():
        repres[expr, cand] = assertrel(expr, cand, "", "")
        repres[cand, expr] = assertrel(cand, expr, "", "")
        if expr == cand:
            return repres
```

Figure 6: `emulate` computes a structured representation of the input string `expr` that preserves any semantic relation $\circ$ in terms of assertion queries. The iterable `all_strings` is defined in Figure 5.

preserve semantic equivalence. While equivalence is an important part of semantics, other semantic relations are also necessary for language understanding. For example, the following feature prominently in theories of linguistic semantics:

- **Entailment** In general terms, an entailment (Winter, 2016) relation $\rightarrow$ is a partial order over Y. Intuitively, if $y \rightarrow y'$, then y is a ‘‘special case’’ of y'. For example, one could construct E, a semantic analysis of English, where $[\![\textit{fat cat} \mid a, \textit{sits}]\!]_E \rightarrow [\![\textit{cat} \mid a, \textit{sits}]\!]_E$.
- **Contrary negation** Negation is a complex topic in semantics. One sense of negation is if two meaning representations are ‘‘contrary’’ (Horn and Wansing, 2020), meaning both cannot be true at the same time.

Does Theorem 2 generalize to other relations besides $=$? To answer this, we first extend assertions and emulation to apply to a generic relation $\circ : M^2$. The proof for Theorem 1 does not fully translate to this new setting, but we will show via a new argument that emulation is still possible.

Definition 7 For $e, e', \in \Sigma^\star$ and $\kappa \in (\Sigma^\star)^2$, define the *assertion oracle*

$$\aleph_{L,\circ}(e, e' \mid \kappa) = \begin{cases} 1 & \text{if } [\![e \mid \kappa]\!]_L \circ [\![e' \mid \kappa]\!]_L \\ 0 & \text{otherwise.} \end{cases}$$

Definition 8 A class of languages $\mathcal{L}$ over Σ is $\aleph$-emulatable w.r.t. $\circ$ if there exists an oracle Turing machine μ and standard Turing machine δ such that, for all $L \in \mathcal{L}$, $\kappa \in (\Sigma^\star)^2$, and $e, e' \in \text{supp}_L(\kappa)$,

$$[\![e \mid \kappa]\!]_L \circ [\![e' \mid \kappa]\!]_L \iff \delta\big(\mu_L(e), \mu_L(e') \mid \kappa\big).$$

We now are ready to prove the extended form of Theorem 1. The main idea of the proof will be to memoize the value of the relation $\circ$ between $[\![e \mid \kappa]\!]_L$ and the values of all expressions smaller than e. This guarantees that δ will be able to ‘‘look up’’ the correct output.

Theorem 5 TRANSPARENT *is* $\aleph$*-emulatable w.r.t.* $\circ$.

Proof. Similarly to Theorem 1, we present the proof constructively as a Python program to

compute μ. We then show how to define δ appropriately, completing the proof.

Figure 6 shows the algorithm to compute $\mu_L(e) \in M$. In Python, $\mu_L(e)$ is a dictionary; we interpret it as a function $\mu_L(e) : \Sigma^\star \times \Sigma^\star \to \{0, 1, \emptyset\}$, where $\emptyset$ represents values that are not set. We define δ as follows:

$$\delta(m, m' \mid \kappa) \iff \begin{cases} m(e, e') & \text{if } m(e, e') \neq \emptyset \\ m'(e, e') & \text{otherwise.} \end{cases}$$

Crucially, it must be that $\mu_L(e)(e, e') \neq \emptyset$ or $\mu_L(e')(e, e') \neq \emptyset$. In Figure 6, `cand` either reaches e before e', or e' before e. By symmetry, assume it reaches e before e'. Then $\mu_L(e')(e, e') \neq \emptyset$, so

$$\begin{aligned} \delta(\mu_L(e), \mu_L(e') \mid \kappa) &\iff \mu_L(e')(e, e') = 1 \\ &\iff \aleph_{L,\circ}(e, e' \mid \lambda^2) = 1 \\ &\iff [\![e \mid \lambda^2]\!] \circ [\![e' \mid \lambda^2]\!] \\ &\iff [\![e \mid \kappa]\!] \circ [\![e' \mid \kappa]\!]. \end{aligned}$$

Therefore `emulate` satisfies Definition 3. □

We needed to change the proof of Theorem 5 compared to Theorem 1 because $\circ$ is not an equivalence relation. In Theorem 1, the final steps relied on reflexivity, transitivity, and symmetry: the three properties that constitute equivalence. The new proof enlarges the size of the emulated representations. Rather than representing each e with a number, $\mu_L(e)$ becomes a large dictionary of strings. This represents an increase in space complexity from linear to exponential in the size of e.

C Old Emulation Definition

A previous version of this paper defined emulation slightly differently. We discuss the differences and explain the advantages of the new definition. First, we defined a *general denotation* as

$$[\![e]\!]_L = \{\langle \kappa, [\![e \mid \kappa]\!]_L \rangle \mid \kappa \in (\Sigma^\star)^2\}.$$

The general meaning represents the meaning of a word across all contexts. Now, say that two functions f, g are isomorphic (with respect to $=$) over a set X iff, for all $x, x' \in X$,

$$f(x) = f(x') \iff g(x) = g(x').$$

We will write $f \cong_= g$ in this case. We will refer to a set of contexts $S \subseteq (\Sigma^\star)^2$ as a *syntactic role*. Each syntactic role has a set of expressions $\text{supp}_L^{-1}(S)$ whose *support* is that role:

$$\begin{aligned} \text{supp}_L(e) &= \{\kappa \in (\Sigma^\star)^2 \mid [\![e \mid \kappa]\!]_L \neq \emptyset\} \\ \text{supp}_L^{-1}(S) &= \{e \in \Sigma^\star \mid \text{supp}_L(e) = S\}. \end{aligned}$$

We can now give the old definition of emulation:

Definition 9 (Old $\aleph$-emulation) $\mu : \Sigma^\star \to M$ emulates $[\![\cdot]\!]_L$ w.r.t. $=$ iff:

1. $\mu \cong_= [\![\cdot]\!]_L$ over $\text{supp}_L^{-1}(S)$, for all $S \subseteq (\Sigma^\star)^2$
2. There exists a Turing machine that computes whether $m = m'$ for each $m, m' \in M$
3. There exists a Turing machine with oracle access to $\aleph_L$ that computes μ

For a set of languages $\mathcal{L}$, this is equivalent to saying $\mathcal{L}$ is $\aleph$-emulatable iff, for all $L \in \mathcal{L}$, there exists an oracle Turing machine μ and normal Turing machine δ such that, for all $S \in (\Sigma^\star)^2$, $e, e' \in \text{supp}_L^{-1}(S)$,

$$[\![e]\!]_L = [\![e']\!]_L \iff \delta\big(\mu_L(e), \mu_L(e')\big).$$

This more closely resembles Definition 3, but we will make two slight changes. First, we will change the quantifier order, such that a single μ must work for every $L \in \mathcal{L}$. Then, we will grant δ access to a context κ, and rephrase the equation to hold over all $\kappa \in (\Sigma^\star)^2$ and $e, e' \in \text{supp}_L(\kappa)$:

$$[\![e \mid \kappa]\!]_L = [\![e' \mid \kappa]\!]_L \iff \delta\big(\mu_L(e), \mu_L(e') \mid \kappa\big).$$

This recovers Definition 3. This version more faithfully reflects the intuitive notion of emulation. The old version required $\mu_L(e)$ to determine how e should evaluate in every possible context. Emulation would not be possible in some cases even with perfect knowledge of L. Now, it must just be possible in any context κ to compute $[\![e \mid \kappa]\!]_L$ from κ and $\mu_L(e)$, which is a weaker standard. Under the new definition, it is *always* possible to emulate a class of languages with one element, assuming $[\![e \mid \kappa]\!]_L$ is computable. An additional improvement is that emulation now applies to all expressions that share a context, whereas before it only targeted expressions with the same support.