



NORTHWESTERN UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-41

September, 2026

Debugging Hand-Curated Natural Language Systems

Constantine Nakos

Abstract

Hand-curated natural language (NL) systems use explicit linguistic knowledge to understand and generate text. This gives them an advantage over pure machine learning (ML) systems in terms of transparency and scrutability: Their knowledge can be inspected, debugged, and improved stably and incrementally over time without the need for retraining. However, maintaining this knowledge can be a challenge. Hand-curated NL systems use specialized forms of knowledge that frequently only a limited number of experts know how to interpret and debug. For a broad-coverage system, the bottleneck is often the time and attention of the few developers who can help it grow.

My thesis addresses this bottleneck by presenting two new techniques that enable non-expert users to debug and extend hand-curated NL systems. The first, Interactive Natural Language Debugging (INLD), concentrates on debugging individual sentences, asking the user simple questions in natural language to drill down on any underlying errors in the system's linguistic knowledge. The second, Generative Example-Driven Error Mining (GEDEM), concentrates on identifying likely errors at scale, asking the user for binary judgments that allow the system to flag potential errors for further review. Together, they reduce the need for expert knowledge to debug the system, instead relying on common linguistic intuition. In doing so, they expand the pool of users who can help maintain a hand-curated NL system.

I ground my discussion in the Companion cognitive architecture and CNLU, its broad-coverage semantic parser. I present implemented INLD and GEDEM systems for CNLU and evaluate them on two sets of examples, examining the reliability of user annotations and the usefulness of the systems' outputs. Lastly, I explore the possibility of automating these systems with large language models (LLMs).

Keywords

Artificial Intelligence, Error Mining, Model-Based Diagnosis, Natural Language Generation, Natural Language Understanding, Reasoning

NORTHWESTERN UNIVERSITY

Debugging Hand-Curated Natural Language Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

For the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Constantine Nakos

EVANSTON, ILLINOIS

September 2026

© Copyright by Constantine Nakos, 2026.

All rights reserved.

Abstract

Hand-curated natural language (NL) systems use explicit linguistic knowledge to understand and generate text. This gives them an advantage over pure machine learning (ML) systems in terms of transparency and scrutability: Their knowledge can be inspected, debugged, and improved stably and incrementally over time without the need for retraining. However, maintaining this knowledge can be a challenge. Hand-curated NL systems use specialized forms of knowledge that frequently only a limited number of experts know how to interpret and debug. For a broad-coverage system, the bottleneck is often the time and attention of the few developers who can help it grow.

My thesis addresses this bottleneck by presenting two new techniques that enable non-expert users to debug and extend hand-curated NL systems. The first, Interactive Natural Language Debugging (INLD), concentrates on debugging individual sentences, asking the user simple questions in natural language to drill down on any underlying errors in the system's linguistic knowledge. The second, Generative Example-Driven Error Mining (GEDEM), concentrates on identifying likely errors at scale, asking the user for binary judgments that allow the system to flag potential errors for further review. Together, they reduce the need for expert knowledge to debug the system, instead relying on common linguistic intuition. In doing so, they expand the pool of users who can help maintain a hand-curated NL system.

I ground my discussion in the Companion cognitive architecture and CNLU, its broad-coverage semantic parser. I present implemented INLD and GEDEM systems for CNLU and evaluate them on two sets of examples, examining the reliability of user annotations and the usefulness of the systems' outputs. Lastly, I explore the possibility of automating these systems with large language models (LLMs).

Acknowledgments

In the garden of graduate school, I've run out of thyme.

I used that same joke for my high school yearbook, which just goes to show you how much my sense of humor has developed since then. My PhD at Northwestern has been one of the most challenging, frustrating, and fulfilling experiences of my life. It has been a long road, and now that I'm at the end, some thanks are in order.

First and foremost, thank you to my advisor Ken Forbus for years of mentorship and collaboration. My time with the Qualitative Reasoning Group was exactly what I needed to grow as a scholar and as a person. You are the reason that happened. You made the sandbox we all play in, the architecture to support our research, and for that I am extremely grateful. The ideas I've developed here will guide me for the rest of my career, and the memories I've made will warm me for the rest of my life. Thank you.

Next, I wish to thank the other guiding lights of QRG: Madeline Usher and Tom Hinrichs. Maddy, you have been a constant source of moral and technical support. Your work is the backbone of this lab, and it's safe to say none of us could do it without you. Tom, you were the on-ramp for generations of QRG students, the one person who knew all the answers. Thank you for years of patient teaching, bug fixes, and acting as the voice of reason we all need.

I would also like to thank the rest of my thesis committee: Dedre Gentner, Ian Horswill, and Kris Hammond. Dedre, you are one of the finest scholars I will ever meet. You've given me a curiosity about analogy and the mind which I plan to pass on to anyone who will listen. Ian, your courses on game development were one of the highlights of my time at Northwestern. Your supportive style of teaching and playful style of research are traits I hope to emulate in my own

career. Kris, your industry, collaborative spirit, and vision for the future of AI are an inspiration. Thank you all for lending me your time and knowledge.

Thanks as well to the Office of Naval Research for funding this research, to my coauthors, and to everyone who helped with my experiments. This thesis would not have been possible without you.

To the members of QRG, past and present, thank you for years of collaboration, commiseration, and companionship. One of the greatest gifts is a good conversation, and you've given me thousands. CJ McFate, you introduced me to Companions and helped instill a love of language that continues to this day. Joe Blass, your sharp wit and questionable sense of humor lit up the lab for years, and I'm proud to consider you a friend. Irina Rabkina, you were the one who showed me how to write a paper, how to teach, and how to succeed in academia, among many other things. Collaborating with you is always a pleasure, and I hope to do so again in the future. Max Crouse, sharing an office with you was a joy. One day you will repay me for all of the gum. Willie Wilson, you introduced me to new areas of research and new methods of teaching. You're a role model and an inspiration. Will Hancock, batting around ideas with you was just as fun as hearing your anecdotes. I hope to one day have half as many adventures as you've had. Sam Hill, every conversation we have leads to something new and interesting. I can't wait to see what you come up with next. Taylor Olson, it's not often you find someone who's on the same wavelength. Thanks for your friendship, for your support, and for listening to all my crazy ideas. Mukundan Kuthalam, I had a blast hacking, hanging out, and brainstorming research ideas with you. Thanks for being a friend. Walker Demel, I've enjoyed my time with you immensely, and I'm proud of the researcher you have become. Showing you the ropes and watching you pass that knowledge on has been one of the most rewarding parts of my time here.

There are many other QRG students who have touched my life in ways too numerous to list here. My deepest thanks to Kezhen Chen, Danilo Neves Ribeiro, Roberto Salas-Damian, Wangcheng Xu, Jiahong Zheng, Zoie Zhao, Xin Lian, and Omar Khater. Thanks as well to the past generations of QRG students who built the systems we use and blazed the trails we walk. And to all of my past professors, advisors, students, teammates, and friends—anyone who has shared a joke, given a word of encouragement, or taken a chance on me—know that you have my thanks.

Finally, heartfelt thanks to my parents George and Debra, my brother David, and my grandmother Eleni. You have supported me in immeasurable ways. I can never hope to repay what you've done. I can only hope to pass some fraction of it on. You are God's greatest gift to me, and I cherish every day we are together.

I love you dearly.

Dedication

For Courtenay, who would have gotten a kick out of all of this.

Table of Contents

Abstract	3
Acknowledgments	4
Dedication	7
Table of Contents	8
Table of Figures	11
Table of Tables	12
1. Introduction.....	13
1.1 Types of NL Systems	14
1.2 Leveraging the User	19
1.3 Claims and Contributions	22
1.4 Road Map.....	23
2. Background.....	24
2.1 Companions	24
2.1.1 Architecture	26
2.1.2 NextKB.....	27
2.1.3 Language Understanding.....	30
2.2 CNLU.....	31
2.2.1 Lexicon.....	34
2.2.2 Grammar.....	35

	9
2.2.3 Semantics.....	40
2.2.4 Valence Patterns	42
2.2.5 Type Checking.....	45
2.2.6 Choice Sets	46
2.3 Verbalize	48
3. Types of Errors	53
4. Interactive Natural Language Debugging.....	58
4.1 The INLD Pipeline.....	59
4.2 Interactive Error Diagnosis.....	63
4.2.1 GDE & CATMS	66
4.2.2 Problem Formulation.....	71
4.3 Debugging Strategies	86
4.4 Evaluation	90
4.4.1 Agreement	93
4.4.2 Analysis	96
4.4.3 Automation.....	98
4.4.4 Discussion	101
5. Generative Example-Driven Error Mining.....	105
5.1 The GEDEM Framework.....	109
5.1.1 The Generative Story.....	110

	10
5.1.2 Generating and Judging Examples	114
5.1.3 Scoring Errors.....	117
5.2 Say.....	120
5.2.1 Setup.....	122
5.2.2 Generating Parse Trees	125
5.2.3 Decoding Output	126
5.2.4 Constraints.....	128
5.2.5 Limitations.....	129
5.3 Evaluation	131
6. Related Work	138
6.1 Model-Based Diagnosis	138
6.2 Interactive Task Learning.....	138
6.3 Explainability.....	139
6.4 Debugging ML Systems.....	140
6.5 Error Mining	142
6.6 Knowledge Engineering.....	143
7. Conclusion	144
7.1 Future Work	145
References.....	148

Table of Figures

Figure 1. The CNLU parsing algorithm.....	33
Figure 2. CNLU lexicon entry for the word “eats”.....	35
Figure 3. Example of a CNLU grammar rule.	37
Figure 4. CNLU noun semtranses for the word “duck.”.....	40
Figure 5. Choice sets for the word “ate” in the sentence “Bob ate the wedge.”.....	42
Figure 6. CNLU verb semtrans for the word “eat.”.....	44
Figure 7. The verbalize algorithm.....	50
Figure 8. Example INLD transcript for the sentence “Bob ate the wedge.”.....	59
Figure 9. The INLD pipeline.....	60
Figure 10. INLD error diagnosis loop.....	71
Figure 11. INLD model-based diagnosis algorithm.....	72
Figure 12. Solve suggestion used for INLD diagnosis.	75
Figure 13. Partial factored interpretations for “the large ducks”.	85
Figure 14. INLD report for “The man became disabled”.	90
Figure 15. Fault correctness by minimum number of annotators who found them.	96
Figure 16. Question about “worried” in “She worried she would catch the flu.”.....	102
Figure 17. CNLU semantics and paraphrases for “The man owns the cat”.	108
Figure 18. Example of eliciting a role filler for the word “laps”.	112
Figure 19. GEDEM generative story for CNLU.....	114
Figure 20. Example of a cluster of GEDEM examples for the word “dodges”.	116
Figure 21. The say generation algorithm.	127
Figure 22. The GEDEM pipeline.....	132

Figure 23. Example of a cluster of GEDEM examples for the word “stigmatizes”. 134

Figure 24. Example of a cluster of GEDEM examples for the word “enrages”. 135

Table of Tables

Table 1. Taxonomy of types of errors found in CNLU. 53

Table 2. Debugging frameworks by input and category of error. 56

1. Introduction

Natural language (NL) systems play an increasingly important role in user interfaces (Wu, et al., 2023), artificial intelligence (AI) applications (Plaat, et al., 2025), and cognitive architectures (Kirk, Wray, Lindes, & Laird, 2022), but their inner workings remain frustratingly opaque. Machine learning (ML) systems use learned representations that are difficult to inspect and debug (Adadi & Berrada, 2018), and their complexity leaves them prone to unreliable behavior (Huang, et al., 2025) and vulnerable to adversarial attacks (Ebrahimi, Rao, Lowd, & Dou, 2017; Zou, et al., 2023). Meanwhile, *hand-curated* NL systems, which are built on explicit linguistic knowledge, can be inspected and debugged, but they struggle to achieve broad coverage, and they require specialized technical expertise to maintain and extend.

Ideally, NL systems should exhibit *transparency* and *scrutability*, properties discussed by Tintarev and Masthoff (2007) in the context of recommender systems but which are equally important for NL. Transparency refers to the user’s ability to understand the inner workings of the system, usually via some form of explanation, while scrutability refers to the user’s ability to correct the system when it is wrong.

ML systems have limited transparency and scrutability. Their learned representations and the details of their behavior are usually too complex for humans to understand. Correcting errors typically happens at a coarse-grained level by gathering new data and retraining the model to improve performance, with no opportunity for the user to fix a specific bug. Efforts in areas such as explainable AI (Adadi & Berrada, 2018) and machine teaching (Simard, et al., 2017) seek to place more power in the user’s hands, but their scope is limited by technical challenges, and most ML-based NL systems do not use them.

In theory, hand-curated systems have an advantage in terms of transparency and scrutability because they represent their linguistic knowledge explicitly. But in practice they are opaque for different reasons. The knowledge they rely on is highly technical, depending on both linguistic theory and the formalisms used by the NL system. The average user cannot understand, let alone debug, a feature-based grammar rule without extensive training, and the problem compounds if the NL system also tries to produce semantics for the text it parses. The knowledge can be inspected and debugged, but only by a select set of experts with specialized knowledge of the system.

My thesis addresses this shortcoming by presenting two techniques that enable non-expert users to debug and extend hand-curated NL systems. The first, Interactive Natural Language Debugging (INLD), concentrates on debugging individual sentences, asking the user simple questions in natural language to drill down on any underlying errors in the system’s linguistic knowledge. The second, Generative Example-Driven Error Mining (GEDEM), concentrates on identifying likely errors at scale, asking the user for binary judgments that allow the system to flag potential errors for further review.

1.1 Types of NL Systems

When discussing natural language systems, it is important to distinguish between *hand-curated* NL systems and NL systems that rely solely on machine learning (ML). Hand-curated NL systems are those that rely on explicit linguistic knowledge which a human is ultimately responsible for maintaining. For example, LEIAs (McShane & Nirenburg, 2021), Rosie (Kirk & Laird, Interactive task learning for simple games, 2014; Mohan, Mininger, Kirk, & Laird, 2012), and Cogent (Galescu, Teng, Allen, & Perera, 2018) all use hand-maintained linguistic knowledge

to convert input text into formal semantic representations in their respective domains. Such systems may have components that are based on ML, and they may use knowledge that is acquired automatically, but at least part of their behavior is determined by a grammar, lexicon, or some other body of linguistic knowledge that a developer must inspect and edit by hand.

Full ML systems, in contrast, rely on linguistic knowledge that is acquired through machine learning and which the developers of the system are not directly responsible for maintaining. Typically, this linguistic knowledge will be encoded in the weights of a neural network (Devlin, Chang, Lee, & Toutanova, 2019; Sutskever, Vinyals, & Le, 2014; Wu, et al., 2023) or the behavior of an embedding space (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013; Pennington, Socher, & Manning, 2014). But even NL systems that produce explicit linguistic knowledge, such as statistically induced grammars (Berant & Liang, 2014), should be viewed as ML systems from the perspective of debugging, as they are generally retrained rather than modified by hand.

Over the past two decades, ML systems have become the dominant paradigm for Natural Language Processing (NLP). From earlier work in text classification with techniques such as Naive Bayes (Hand & Yu, 2001) and Support Vector Machines (Cortes & Vapnik, 1995) to word embedding methods such as word2vec (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013) and GloVe (Pennington, Socher, & Manning, 2014) to more sophisticated neural models such as LSTM-powered sequence-to-sequence generation (Dušek, Novikova, & Rieser, 2018), ML systems have benefited from the development of larger and larger datasets, improved computing power, and an approach that minimizes the amount of language expertise required to train a system. More recently, the Transformer architecture (Vaswani, et al., 2017) has facilitated the creation of large language models (LLMs) that exhibit a wide range of linguistic capabilities,

such as in-context (Brown, et al., 2020) and zero-shot learning (Kojima, Gu, Reid, Matsuo, & Iwasawa, 2022), instruction-following (Ouyang, et al., 2022), and tool use (Schick, et al., 2023).

This power comes with a price. Large ML models, especially neural networks, are difficult to inspect. No user, even a highly trained developer, can fully understand the behavior of a model with millions or billions of parameters. Instead, developers rely on large test sets to gauge a model’s performance over thousands of examples. If there is a specific type of input a ML model is having trouble with, the developers can construct more targeted test sets with tools like CheckList (Ribeiro, Wu, Guestrin, & Singh, 2020) to ensure the adequacy of its output. But in both cases, the correction is coarse-grained, focusing on improving the model over a set of examples rather than attempting to understand and debug individual errors.

Explainable AI (Adadi & Berrada, 2018) techniques offer more tools for understanding the behavior of a ML model, but they are still limited in terms of detail and fidelity. Tools like LIME (Ribeiro, Singh, & Guestrin, 2016) allow users to visualize the decision boundary of a classifier, while other techniques such as SHAP (Lundberg & Lee, 2017) quantify the effects of different features on a decision. Probing (Ettinger, 2020) can shed light on some of the model’s behaviors, as can training a more interpretable model to replicate the target model’s behavior in parallel (Lindsey, et al., 2025). But these techniques abstract away the true complexity of the model, making them at best an approximation of its behavior.

The gap between user-accessible explanations and the true complexity of models is most apparent in LLMs. Autoregressive LLMs often *hallucinate* (Huang, et al., 2025) plausible sounding but erroneous answers to user queries. They are trained to predict the most probable next token, and there is no guarantee that this token will be “true” in any concrete sense. LLMs are also sensitive to slight changes in input, leaving them vulnerable to adversarial attacks (Zou,

et al., 2023) that use targeted changes to the input to manipulate the output the LLM produces. Taken together, this means that the behavior of LLMs is very difficult to pin down. Even if one prompt produces the correct answer, there is no way to guarantee that a slight variation of the prompt won't produce a hallucination or malicious output.

To make matters worse, LLMs produce fluent explanations that have nothing to do with their actual reasoning (Lindsey, et al., 2025). At best, these explanations reflect the reasoning steps a human might use to reach the same conclusion. At worst, they can actively mislead the user as to how the LLM produced its answer. Thus, LLMs are not only opaque but deceptive, making them a poor fit for applications where transparency, stability, and precision are important.

Finally, ML-based NL models are difficult to correct. Tools such as CheckList (Ribeiro, Wu, Guestrin, & Singh, 2020) and AdaTest (Ribeiro & Lundberg, 2022) help developers generate test sets to perform targeted evaluations, which can be used to shore up weaknesses in the model through additional training. But the fix for one set of bugs may re-introduce another set that had previously been fixed, requiring iteration to stabilize, and developers can only address bugs indirectly by creating more training data. Thus, models can usually only be corrected through coarse-grained training or retraining, and the end user has little opportunity to correct the behavior of the system directly.

Hand-curated NL systems, in contrast, have the potential to be transparent and scrutable, but they are limited by the complexity of their representations. An expert in the system can trace through its behavior on a particular input, identify where it goes wrong, and propose a fix to the system's linguistic knowledge that will remedy the problem. Given a trained expert, one or two

illustrative examples, and enough time, any error in a hand-curated NL system can be fixed, subject to the limits of the system’s algorithms and representations.¹

But the number of experts in any particular system is low, and the effort required to fix a single bug ranges from negligible to immense. Thus, hand-curated NL systems are subject to the *knowledge acquisition bottleneck* (Olson & Rueter, 1987), wherein the performance of the system is limited by its ability to acquire knowledge—or, in this case, correct it. The more complicated and specialized the system, the tighter the bottleneck.

Furthermore, improvements to one hand-curated system do not readily transfer to others. Linguistic resources such as WordNet (Miller, 1995), VerbNet (Schuler, 2005), and FrameNet (Fillmore, Wooters, & Baker, 2001) have played roles in multiple, independent NL systems, but formalism-dependent knowledge such as grammar rules is often limited to a single system or family of systems. Whereas almost any ML-based NL system can benefit from a new corpus, hand-curated NL systems are siloed by the specialized types of knowledge they require. Even worse, the knowledge requirements may evolve over time as the developers of the system adjust its formalisms to handle new cases.

Even with these limitations, hand-curated NL systems have properties that make them attractive for certain applications. They are robust to the kinds of perturbations that LLMs struggle with. Slight changes to the input might hurt the system’s understanding of it, but they will not cause the system to hallucinate something new altogether. When errors do occur, they can be traced to their source and fixed. Moreover, most bugs stem from issues with particular words or linguistic constructs, so the fixes are local and incremental. Once fixed, these bugs are unlikely to resurface. Because the systems are not retrained from scratch, developers can

¹ For example, a context-free grammar cannot properly handle linguistic constructs that depend on context.

gradually improve their behavior over time. For example, sustained effort from the head-driven phrase structure grammar (Pollard & Sag, 1994) and combinatory categorial grammar (Steedman, 2001) communities has led to the creation of robust syntactic parsers using broad-coverage grammars written and maintained by hand.

1.2 Leveraging the User

My approach to relieving the knowledge acquisition bottleneck for hand-curated NL systems stems from the following observation: Users know a lot about language, even if they know very little about a system’s internal workings. The system’s knowledge is complementary. It has, or can be given, an understanding of its internal workings, but it has no way to tell when its understanding of language is wrong. When the user gives the system a sentence to parse, the user knows what the sentence means, but not necessarily how to represent it using the system’s internal representations. Meanwhile, the system has access to its representations but has no independent understanding of language or the world to check itself against.

This suggests a natural symbiosis: If the system can tap into the user’s knowledge by asking targeted questions about a sentence, it can use the answers to diagnose and correct errors in its own linguistic knowledge. The user acts as an oracle—an expert in everyday language—while the system poses questions to drive its reasoning about itself, eliminating hypotheses until it reaches a conclusion about whether and how its knowledge is buggy.

In my thesis, I explore this idea in two ways. Interactive Natural Language Debugging (INLD) applies this idea to individual sentences. It casts the behavior of a hand-curated NL system as a reasoning problem, then determines which questions to ask the user to drill down on potential errors. This process is iterative. With enough questions, the system will be able to

localize the error in its knowledge, if any such error exists, and, in some cases, ask follow-up questions to construct a fix. Notably, all of the questions the user must answer are posed in natural language and require no system-specific expertise. Thus, INLD both reduces the amount of effort needed to fix bugs in a hand-curated NL system and opens debugging up to non-expert users, making it possible for the system’s linguistic knowledge to be maintained and extended at scale.

As a complement to this technique, Generative Example-Driven Error Mining (GEDEM) asks users simple questions to locate potential bugs in the system’s linguistic knowledge. Where INLD focuses on specific sentences, GEDEM operates over batches of sentences. Specifically, it samples semantics from the system’s space of semantic representations and uses its linguistic knowledge to generate a group of sentences meant to convey the same meaning. GEDEM then asks the user if any of the generated sentences are ungrammatical or fail to convey the intended meaning. These judgments are then aggregated across sentences and used to determine which pieces of linguistic knowledge are likely to be responsible for the observed errors.

I ground my discussion of these techniques in CNLU (Tomai & Forbus, 2009), the natural language understanding module of the Companion cognitive architecture (Forbus & Hinrich, 2017). CNLU is an ideal test bed for INLD and GEDEM because it has broad but imperfect coverage, it relies on several types of explicit linguistic knowledge that require expertise to debug, and it already has strong ties to FIRE (Forbus K. D., Hinrichs, de Kleer, & Usher, 2010), Companions’ reasoning engine, and NextKB², a general ontology that supports knowledge representation and reasoning. Taken together, these factors make CNLU a hand-

² <https://www.qrg.northwestern.edu/nextkb/index.html>

curated NL system that both needs INLD and GEDEM and has the existing resources to support them.

Although I focus on CNLU in this thesis, it is important to note that the INLD and GEDEM frameworks can also apply to other hand-curated NL systems. The only hard requirements are that the system has explicit linguistic knowledge that can be debugged and that its parses can be formulated as a reasoning problem. In addition, GEDEM requires that the NL system be able to generate text as well as parse it.

While purely syntactic systems meet these requirements, INLD and GEDEM best fit NL systems that handle semantics. They are especially well-suited to the NL components of cognitive architectures, such as LEIAs (McShane & Nirenburg, 2021) or Soar (Laird, et al., 2017), which already have advanced knowledge representation and reasoning capabilities but typically run into issues with maintenance and coverage.

INLD and GEDEM also fit nicely with the goal of lifelong learning (Chen & Liu, 2018), in which a software agent continues to learn and update its capabilities over a sustained period of time. Additionally, they can help ensure that a grammar has appropriate coverage for a specific domain, such as game-playing (Kirk & Laird, Interactive task learning for simple games, 2014), collaborative problem-solving (Galescu, Teng, Allen, & Perera, 2018), or drawing (Wang, Ginn, Liang, & Manning, 2017).

Finally, it is worth noting that there are limits to this approach. INLD and GEDEM are best suited to errors that require changing specific facts, such as the meaning of a word, rather than errors that require redesigning part of the system. This is because larger changes often require system-specific expertise and conscious design decisions that are not amenable to automated reasoning. In other words, INLD and GEDEM explore the space already set forth by

the system developers. They can fix errors, patch gaps in coverage, and point to more serious issues for the developers of the system to investigate, but as they currently stand, they cannot automate the entire development process.

1.3 Claims and Contributions

In this thesis, I put forth the following claims:

1) Casting the behavior of a hand-curated NL system as a reasoning problem can allow it to diagnose errors by asking a non-expert user targeted questions in natural language.

2) Likely errors in a hand-curated NL system’s linguistic knowledge can be identified by asking non-expert users to judge sentence paraphrases generated by the system.

In support of Claim 1, I make the following contributions:

1.1) I present Interactive Natural Language Debugging, a framework for diagnosing and correcting errors in hand-curated NL systems by reasoning about the system’s behavior and asking users diagnostic questions in natural language.

1.2) I present an implemented INLD system for CNLU (Tomai & Forbus, 2009), the natural language understanding system for the Companion cognitive architecture, and evaluate its utility on data from an NL corpus.

In support of Claim 2, I make the following contributions:

2.1) I present Generative Example-Driven Error Mining, a framework for identifying likely errors in a hand-curated NL system’s linguistic knowledge by generating paraphrases and eliciting judgments from non-expert users.

2.2) I present an implemented GEDEM system for CNLU and evaluate its utility by identifying errors in a sample of CNLU’s semantic knowledge.

1.4 Road Map

The remainder of my thesis is structured as follows. I begin by reviewing the Companion cognitive architecture (Section 2.1), its natural language understanding system CNLU (Section 2.2), and its template-based natural language generation utility verbalize (Section 2.3). I then discuss the types of errors that can arise in CNLU and how they can be addressed (Section 3).

Next, I present the INLD pipeline (Section 4.1), how error diagnosis can be implemented as a model-based diagnosis problem (Section 4.2), debugging strategies for CNLU (Section 4.3), and evaluation on sentences taken from a corpus of stories (Section 4.4).

Pivoting from individual sentences to batches, I next present the GEDEM framework (Section 5.1), an extension of error mining that can find errors in semantic parsers. I then discuss say (Section 5.2), a new NLG system based on the linguistic knowledge and parsing algorithm of CNLU. I use say to implement a GEDEM system to debug CNLU’s linguistic knowledge and evaluate its performance over a corpus of 209 generated examples (Section 5.3).

I wrap up by reviewing a range of related work, including work on model-based diagnosis (Section 6.1), Interactive Task Learning (Section 6.2), explainability (Section 6.3), debugging machine learning-based NL systems (Section 6.4), error mining in hand-curated NL systems (Section 6.5), and knowledge engineering (Section 6.6). Finally, I conclude and discuss future work (Section 7).

2. Background

Before discussing INLD and GEDEM, it will be useful to review the system I will be applying them to. I begin with an overview of the Companion cognitive architecture (Section 2.1), whose reasoning capabilities and need for broad-coverage semantic parsing make it an ideal test bed for interactive debugging. I then describe the workings of CNLU (Section 2.2), Companions' natural language understanding (NLU) module, to ground further discussion in the specific processes and representations CNLU uses. Finally, I present *verbalize* (Section 2.3), a template-based natural language generation (NLG) utility that allows a Companion's knowledge to be expressed in natural language. Taken together, these three systems form the foundation for INLD and GEDEM.

2.1 Companions

The Companion cognitive architecture (Forbus & Hinrich, 2017) seeks to understand cognition by building *software social organisms*, AI systems that can reason, learn, and interact the way that people do. Prerequisites to this goal include a rich knowledge base containing a breadth of commonsense knowledge, the ability to interact through modalities that are natural to people, and a suite of methods for reasoning, planning, and learning to drive intelligent behavior. In particular, the architecture places an emphasis on qualitative representations (Crouse, McFate, & Forbus, 2018; Forbus K. D., 2019; Hinrichs & Forbus, 2015), a human-like way to reduce the complexity of a continuous world, and analogical reasoning (Forbus, Gentner, & Law, 1995; Forbus, Ferguson, Lovett, & Gentner, 2017; McLure, Friedman, & Forbus, 2010), a data-efficient and psychologically plausible method of learning from past experience.

While the Companion cognitive architecture continues to grow and develop, it has already provided the basis for numerous applications and cognitive models. The CogSketch sketch understanding system (Forbus, Usher, Lovett, Lockwood, & Wetzel, 2011) has been used to model geometric analogies (Lovett, Tomai, Forbus, & Usher, 2009), physics problems (Chang, Wetzel, & Forbus, 2011), and diagrams from scientific texts (Xu & Forbus, 2025). It has also been used as an educational tool in several courses (Forbus, Chang, McLure, & Usher, 2017; Forbus, et al., 2018); its sketch worksheets allow students to draw diagrams and receive interactive feedback by comparison to a reference diagram provided by the instructor.

On the language side, CNLU (Tomai & Forbus, 2009) augmented with analogical training (Crouse, 2021) serves as the backbone for a question-answering information kiosk (Wilson, et al., 2019) deployed in the Northwestern University Computer Science Department. More recently, the SocialBot project (Nakos & Forbus, 2025) has extended this functionality and made it available online. The SocialBot answers questions about classes, people, and events and tracks user preferences about event topics and food offerings. Companions have also been used to model how people acquire linguistic constructions (McFate C. J., 2018), resolve referring expressions (Nakos, Rabkina, & Forbus, 2020; Nakos, Rabkina, Hill, & Forbus, 2020), and reason about legal cases (Blass J. A., 2023). Other research has shown how Companions can learn commonsense knowledge (Barbella D. M., 2016; Ribeiro & Forbus, 2021) and qualitative models (Crouse, McFate, & Forbus, 2018; McFate, Forbus, & Hinrichs, 2014) from text.

Finally, Companions have been used to model human-like learning and reasoning in a variety of domains. Analogical Theory of Mind (Rabkina, 2020) models how people reason about each other's beliefs. It not only fits human experimental data well (Rabkina, McFate, Forbus, & Hoyos, 2017) but is more robust to missing information and noise than comparable

plan recognition systems (Rabkina, Kantharaju, Wilson, Roberts, & Hiatt, 2022). Defeasible Deontic Inheritance Calculus (Olson T. , 2025) models how people resolve conflicts between norms, which is important for resolving conflicting testimony, while MoralDM (Dehghani, 2009) models moral decision-making across cultural contexts. Other work shows how qualitative models can be acquired (Hinrichs & Forbus, 2012; Hinrichs & Forbus, 2019) and applied (Forbus & Hinrichs, 2019; Hinrichs & Forbus, 2015) in the context of strategy games, and how these models can be used to partition time and space (Hancock, 2025).

In short, the Companion cognitive architecture uses a broad knowledge base, qualitative representations, and an ensemble of reasoning and learning techniques to model human cognition and perform tasks across many domains. In the following subsections, I briefly discuss the architecture itself (Section 2.1.1), its knowledge base (Section 2.1.2), and its need for a robust NLU system (Section 2.1.3).

2.1.1 *Architecture*

The Companion cognitive architecture consists of a set of *agents* that operate in parallel. Each agent has a specific responsibility, such as reasoning through problems (the Session Reasoner), interacting with the user in natural language (the Interaction Manager), managing the Companion session (the Facilitator), or keeping knowledge in sync between agents (the Archivist). Companion agents can be located on the same computer, in which case they may share a knowledge base (KB), or they can be run on separate computers to avoid contention for resources.

Agents use Hierarchical Task Networks (Nau, Cao, Lotem, & Munoz-Avila, 1999) to plan their actions. They work together using coarse-grained parallelism, coordinating their behavior at a high level while leaving the specifics of a given task to each individual agent. This

architecture supports modularity at both the code and knowledge levels and allows Companions to be configured for different domains without requiring extensive changes.

Most Companion agents are equipped with an instance of the FIRE reasoning engine (Forbus K. D., Hinrichs, de Kleer, & Usher, 2010). FIRE is a robust, flexible, and efficient reasoning engine that supports a variety of reasoning techniques. It provides an interface to the agent's instance of the KB, maintains a working memory (WM) of cached beliefs using an LTRE (Everett & Forbus, 1996), and supports the definition of *outsourced predicates*, procedural attachments which make specialized, code-level operations available from the knowledge level during reasoning. Other capabilities of note include a Horn clause-driven backchaining algorithm called *query*; an incremental and controllable problem solver that uses AND-OR graphs called *solve* (Paritosh & Forbus, 2005); algorithms for analogical mapping (Forbus, Ferguson, Lovett, & Gentner, 2017), retrieval (Forbus, Gentner, & Law, 1995), and generalization (Kandaswamy, Forbus, & Gentner, 2014; McLure, Friedman, & Forbus, 2010); and the SHOP-based (Nau, Cao, Lotem, & Munoz-Avila, 1999) planner used to plan agent actions.

2.1.2 *NextKB*

Among FIRE's many responsibilities, perhaps the most important is providing access to NextKB³, the knowledge base used by Companions. NextKB represents facts in predicate calculus using the CycL language (Lenat & Guha, 1991). Its contents are derived in part from OpenCyc, an open-source subset of the Cyc⁴ knowledge base, and they have been augmented

³ <https://www.qrg.northwestern.edu/nextkb/index.html>

⁴ <https://cyc.com/>

with Companion-specific plans and reasoning rules, better support for visual, spatial, and other qualitative representations, and a wide range of linguistic knowledge.

Concepts are represented by *constants* like `Kimble-TheCat` and *non-atomic terms* like `(FruitFn AppleTree)`, where `FruitFn` is a *logical function* that denotes the fruit of the concept that is its argument. Concepts can be *individuals*, representing specific entities, or *collections*, representing types of entities. Individuals are mapped to collections with the `isa` predicate: `(isa Kimble-TheCat Cat)` declares that Kimble is a cat. The `genls` predicate indicates that one collection is a generalization of another. For example, `(genls Cat Mammal)` indicates that cats are a kind of mammal, and that all instances of the collection `Cat` are also instances of the collection `Mammal`. `genls` is transitive, so `(genls Mammal Animal)` would imply that all cats are animals.

Facts are represented by a *predicate* and an appropriate number of arguments. For example, `(eatsWillingly Kimble-TheCat Chicken-Meat)` represents the fact that Kimble willingly eats chicken. The *arity* of a predicate indicates how many arguments it should take, typically between one and seven. `genlPreds` indicates that one predicate entails another. For example, `(genlPreds touches near)` states that if A is touching B, they are also near each other, or that `(touches A B)` entails `(near A B)`.

`argIsa` relations specify what types the arguments of a predicate should have. For example, `(argIsa eatsWillingly 1 Animal)` states that the first argument of the `eatsWillingly` relation must be an instance of `Animal`. `argIsa` constraints are useful for checking the integrity of generated knowledge. For example, they allow CNLU to eliminate nonsensical interpretations of a sentence while parsing (i.e., interpretations that produce statements which violate a predicate's type constraints).

Note that, by convention, individuals and collections are capitalized (*Kimble-TheCat*, *Chicken-Meat*), predicates start with a lower-case letter (*eatsWillingly*), and functions are capitalized and have the suffix “Fn” (*FruitFn*).

Facts in NextKB are stored in one or more *microtheories* (Guha, 1992). Microtheories are bundles of logically consistent and conceptually related facts that provide contexts for reasoning. FIRE reasoning operations, including KB retrievals, are contextualized with a microtheory that limits which facts are visible. In this way, conflicting models, fictional worlds, and hypothetical situations can be kept separate from each other and only accessed when specifically desired, preventing crosstalk and reducing the number of irrelevant rules and facts that must be considered while reasoning.

One microtheory can inherit from another via the *genlMt* predicate. For example, (*genlMt WorldHistoryMt UnitedStatesHistoryMt*) indicates that facts about United States history should be considered when reasoning about world history. *genlMt* is transitive, making it a simple and flexible way to aggregate separate bundles of knowledge for reasoning.

NextKB contains over 83,000 collections, 26,000 predicates, and 700,00 facts distributed across 1,300 microtheories. Among these facts are 191,000 lexicon entries that specify the grammatical features of words, 70,000 *semtranses* (“semantic translations”) that map words to their meanings, and 467 grammar rules. These facts form the basis of Companions’ language understanding capabilities.

The knowledge in NextKB is hand curated. The bulk of it was written by hand, either as part of OpenCyc or as an extension of it. In some cases, such as the lexicon and *semtranses*, the facts were compiled semi-automatically from an existing resource, then corrected by hand as

needed. While hand curation by experts has been adequate for Companions so far, it remains a bottleneck for knowledge-intensive tasks and a source of errors when the developers make mistakes. My research focuses on testing and correcting linguistic knowledge specifically, but similar techniques could be used to test other knowledge in NextKB. In general, we are working towards the goal of *knowledge autonomy* (Nakos & Forbus, 2025), where a Companion has the tools it needs to acquire and manage knowledge on its own, without direct developer intervention.

2.1.3 *Language Understanding*

NextKB provides a rich substrate for language understanding. The semantics of a piece of text can be represented with CycL sentences using concepts from the KB, while reasoning rules can assist in the interpretation of the text and adapt it for specific domains (McFate, Forbus, & Hinrichs, 2014; Tomai E. R., 2009). Thus, NextKB serves two roles in language understanding: as a source of linguistic knowledge that can be used to understand some piece of text (grammar rules, lexicon entries, etc.), and as a source of concepts with which to represent the text’s meaning (via concepts and predicates).

Language has proven to be very important for Companions. It is a natural modality for interacting with people, as seen in the SocialBot project (Nakos & Forbus, 2025; Wilson, et al., 2019) and game learning (Hinrichs & Forbus, 2014), making it a crucial capability for developing software social organisms. It is also an excellent way to acquire knowledge. Past work has shown how a Companion can acquire knowledge from science texts (Barbella D. M., 2016; Ribeiro, et al., 2019), Simple English Wikipedia⁵ articles (Ribeiro D. N., 2023), legal case

⁵ https://simple.wikipedia.org/wiki/Simple_English_Wikipedia

summaries (Blass J. A., 2023), and user-provided microstories (Blass & Forbus, 2017). Learning by reading is one way to sidestep the knowledge acquisition bottleneck, reduce Companions’ reliance on hand-curated knowledge, and achieve knowledge autonomy. It shifts the maintenance burden from all of the general knowledge in the KB to the linguistic knowledge that can be used to acquire new knowledge. Put another way, having a robust means of language understanding—and a way to debug the knowledge required—makes it easier for Companions to learn on their own at scale.

2.2 CNLU

Companions Natural Language Understanding (Tomai & Forbus, 2009), or CNLU, is the NL component of the Companion architecture. Its primary function is to take a passage of English text, parse it, interpret it semantically, and install the interpretation in the working memory of a FIRE reasoner for downstream use.

While CNLU can be invoked on its own, it is typically used by a Companion’s Interaction Manager agent, which keeps track of the linguistic input and output for a Companion session. Thus, when a user asks the Companion a question in natural language, the Interaction Manager will invoke CNLU to interpret the question, use that interpretation to perform the reasoning needed to formulate an answer, and respond. Likewise, when a Companion is reading a piece of text, the Interaction Manager will use CNLU to interpret the passage one sentence at a time, then save off any interesting knowledge it gleaned from the interpretations.

CNLU uses a rule-based, bottom-up chart parser based on the TRAINS parser (Allen, 1994). The core parsing algorithm *tokenizes* the input, constructs *constituents* that represent the syntactic and semantic properties of each token, and incrementally adds them to a *chart*, a data

structure which maps spans of tokens in the input to their corresponding constituents. The parser uses a context-free, feature-based grammar to construct new constituents and build up parse trees. Whenever a new constituent is added to the chart, the algorithm walks backwards to see if it can be used to match the *right-hand side* of a grammar rule. If the algorithm finds a path of contiguous constituents that match a rule, it instantiates a new constituent covering the whole span based on the *left-hand side* of a rule (e.g., combining a determiner and a noun to create a noun phrase). It then places this constituent on an *agenda*, along with any other constituents that haven't been processed yet. One by one, constituents are popped from the agenda and added to the chart, possibly completing more rules. The cycle continues until all input tokens have been processed and the agenda is empty, meaning no more work can be done. At that point, the top parse trees (i.e., the constituents that best cover the input tokens) are installed in FIRE's working memory. Pseudocode for this algorithm can be seen in Figure 1. (Parsing constraints are discussed in Section 5.2.4.)

One of the capabilities that sets CNLU apart from other parsers is that it builds up semantic interpretations as well as syntactic ones. The grammar rules have features that specify what a new constituent's semantics should be based on the semantics of the child constituents used to instantiate it. The procedure for updating the semantics is somewhat complex, but the net result is that every parse tree produced by CNLU has a semantic interpretation. The need for high-precision semantic understanding grounded in NextKB is the main reason Companions rely on CNLU for language understanding rather than an off-the-shelf syntactic parser.

CNLU does not just produce a single interpretation for a piece of text, but rather a space of them. Language is inherently ambiguous, so the same sentence can have multiple meanings depending on context. CNLU attempts to capture as many of these meanings as possible so the

```

// Given sentence s, parsing constraints Cn:
// Parse s and install its interpretations in working memory.
Function PARSE(s, Cn):
  A = MAKE-AGENDA() // The agenda, a priority queue.
  Ch = MAKE-CHART() // The chart, an array.
  G = RETRIEVE-GRAMMAR() // The grammar, a set of rules.
  T = TOKENIZE(s) // Token input queue.
  // Parse until out of constituents to process.
  Until EMPTY(T) and EMPTY(A):
    If EMPTY(A):
      // Empty agenda, so process the next input token.
       $t_i$  = DEQUEUE(T)
      // Retrieve all definitions in the lexicon for token  $t_i$ .
      For d in RETRIEVE-DEFINITIONS( $t_i$ ):
        // Make constituent for definition and add to the agenda.
        c = MAKE-CONSTITUENT(d, i, i+1)
        ENQUEUE(A, c)
    Else:
      // Process the next constituent on the agenda.
      c = DEQUEUE(A)
      // Only add c to the chart if it satisfies any parsing constraints.
      If CHECK-CONSTRAINTS(c, Cn):
        INSERT(Ch, c)
        // Check every rule that could be terminated by c.
        For r in APPLICABLE-RULES(G, c):
          // Walk the chart for constituent sequences ending in c.
          For seq in APPLICABLE-SEQUENCES(Ch, r, c):
            // Try to apply r to seq to make a new constituent.
             $c_{new}$  = APPLY-RULE(r, seq)
            If  $c_{new}$ :
              // Add the new constituent to the agenda.
              ENQUEUE(A,  $c_{new}$ )
  // Extract top-level parse trees and install them in working memory.
  For pt in BEST-PARSE-TREES(Ch):
    INSTALL-IN-WM(pt)

```

Figure 1. The CNLU parsing algorithm.

downstream reasoning task can choose whichever one is most appropriate. Syntactically, CNLU can produce multiple parse trees for the same sentence, which correspond to different parts of speech and/or sentence structures. To use a classic example, CNLU would produce different parse trees for “I [[saw the man] [with the telescope]]” and “I [saw [the man with the telescope]]”. Semantically, CNLU maintains *choice sets* corresponding to different

interpretations of a token in the input. For example, in the sentence “Mary had a little lamb,” CNLU would produce a choice set for the word “had” containing choices for owning, giving birth, and eating, all of which are valid interpretations of the word in the right context.

While this overview covers the behavior of CNLU at a high level, for INLD and GEDEM, it will be helpful to discuss CNLU in more detail, in particular the types of knowledge it uses. I begin with the lexicon (Section 2.2.1) before moving on to the grammar (Section 2.2.2), semantics (Section 2.2.3), valence patterns (Section 2.2.4), type checking (Section 2.2.5), and choice sets (Section 2.2.6).

2.2.1 *Lexicon*

The lowest layer of CNLU’s language understanding is a broad-coverage *lexicon* that maps input tokens to their root forms, parts of speech, and grammatical features. Whenever the parsing algorithm encounters a new input token, it looks up the corresponding lexicon entries and uses them to instantiate a set of constituents to add to the chart. For example, the word “ducks” has two lexicon entries: a plural noun and a third-person singular verb. If the algorithm encounters “ducks” in the input, it will instantiate two constituents at that position, one for the noun and one for the verb. One or both of these constituents might make it into a top-level parse tree, depending on the grammar rules and the surrounding tokens (e.g., “the girl ducks” vs. “ducks eat bread”).

The lexicon consists of 191,000 lexicon entries across 15 parts of speech. This includes content words like nouns, verbs, adverbs, and adjectives, and function words like prepositions and determiners. Note that the lexicon only concerns itself with morphology, parts of speech, and grammatical features. Word semantics are stored separately.

For example, Figure 2 shows the lexicon entry for the word “eats”, stored in NextKB as a `definitionInDictionary` statement. The root form is `Eat-TheWord` (NextKB’s concept for the word “eat”), the part of speech is `Verb`, and the final argument lists the grammatical features an “eats” constituent should be instantiated with. `(vform (TheSet pres))` indicates that the verb form of “eats” is present tense. `(agr (TheSet 3s))` indicates that it is the third-person singular form of the verb, so it should only match third-person singular subjects (“the duck eats” but not “the ducks eats”). The `subcat` (“subcategorization”) feature controls what syntactic arguments the verb can take, in this case various configurations of a noun phrase (the direct object) and a prepositional phrase. The lexicon tracks 43 features, some of which are used by the grammar (e.g., `agr` constrains subject-verb agreement) and others which are just used for bookkeeping (e.g., the `root` feature notes the root form of a token).

```
(definitionInDictionary Nulex eats Eat-TheWord Verb
  (TheSet (root eat) (vform (TheSet pres)) (agr (TheSet 3s))
    (subcat (TheSet np np-pp pp pp-conative))))
```

Figure 2. CNLU lexicon entry for the word “eats”.

2.2.2 Grammar

The main purpose of the grammar is to build up larger constituents from smaller ones, filling out the chart and ultimately constructing one or more parse trees for the input sentence. Each constituent consists of a *lexical category* (a part of speech or phrase type), a set of grammatical features, and pointers back to the child constituents it was constructed from and the grammar rule used to make it. The chart also keeps track of the constituent’s location in the sentence (the span of tokens it covers) and its probability (the product of the probabilities of its children and the probability assigned to the grammar used to combine them).⁶

⁶ While probabilities play an important role in other parsing algorithms, they play an ancillary one in CNLU. Rule probabilities are set manually, not learned, and they are mainly used to disfavor uncommon or strained

Lexicon entries are instantiated as *leaf* or *terminal* constituents, which grammar rules can build up into larger constituents. The grammatical features of a leaf constituent are taken from the `definitionInDictionary` statement and supplemented with a `var` feature and a `sem` feature.

The `var` feature contains a *discourse variable*, a unique symbol used to represent the entity that token denotes. For example, the word “eats” might be assigned a discourse variable of `eat4200`, which can then be used to construct predicate calculus statements about the event “eats” refers to. Non-terminal constituents also have `var` features, usually corresponding to the `var` feature of one of their children. So a parse tree for the sentence “the duck eats bread” might have a `var` of `eat4200`, indicating that the sentence is “about” the eating event denoted by “eats”. “the duck” and “bread” would have their own discourse variables as well.

The `sem` feature contains the predicate calculus which represents the semantics of the constituent. These are instantiated using discourse variables, producing semantic expressions like `(isa eat4200 EatingEvent)`. The parsing algorithm processes the `sem` feature specially to handle quantifiers and conjunctions, update choice sets, and substitute in discourse variables when required by a grammar rule (discussed further below).

Grammar rules consist of a rule ID, a left-hand side, and a right-hand side. The left-hand side is a template for constructing a new constituent from a sequence of child constituents that match the right-hand side. For example, Figure 3 shows the grammar rule `-np->det-cnp-`, which builds a `np` (a noun phrase) from a `Determiner` (like “the”) and a `cnp` (a common

grammatical constructions. The parsing algorithm (1) drops constituents whose probability drops below a certain threshold, and (2) when it finds too many parse trees for a sentence, it only installs in WM the ones with the highest probability. But aside from these filters, the algorithm is exhaustive, so even though the agenda considers the constituents in order of decreasing probability, it does not affect what work is done. This differs from parsing algorithms based on beam search, where a desired constituent with lower probability might fall out of the beam.

noun phrase, sans determiner, such as “duck”, “bread”, or “fresh apple”). Its rule ID is boxed in red, its left-hand side (the template of the np to instantiate) is boxed in green, and its right-hand side (the patterns of `Determiner` and `cnp` to match) is boxed in orange.

```
(grammarRule -np->det-cnp-
  (EATermFn np
    (TheList (var ?varcnp) (agr ?a) (det ?vardet) (:NOUN ?varcnp)
      (sem (thereExists ?varcnp ?semcnp))))
  (TheList
    (EATermFn Determiner
      (TheList (agr ?a) (var ?vardet) (sem ?semdet)
        (lex (? !l every no neither what which)) (tposs -)))
    (EAHeadFn
      (EATermFn cnp
        (TheList (agr (? a 1s 2s 3s)) (count -) (var ?varcnp)
          (sem ?semcnp))))))
```

Figure 3. Example of a CNLU grammar rule.

When attempting to apply a rule, the parsing algorithm first searches for sequences of constituents that match the lexical category of the right-hand side. If `-np->det-cnp-` is applied to “the duck”, it might find a `Determiner` constituent for “the”, constructed from the lexicon entry, and a `cnp` constituent for “duck”, constructed from a terminal `Noun` constituent by a different grammar rule.

Once it has a candidate sequence of constituents, the algorithm performs *feature-based unification* to complete the match. For each term in the right-hand side of the rule, its features are matched to the features of the corresponding constituent in the chart, and any *parser variables* are bound accordingly. Thus, the `?varcnp` variable in the rule will be bound to the value of the `var` feature in the `cnp` constituent, which will be a discourse variable like `duck4150`.

Parser variables can also take sets of values. For example, the token “hit” corresponds to four different verb forms: the base form (“to hit”), present tense (“now they hit”), past tense (“once he hit”), and past participle (“we had been hit”). When a constituent is created for “hit”, its `vform` feature will contain a parser variable with these four values, some of which may be filtered out depending on which grammar rules it matches against.

As another example, consider the `agr` feature of the `cnp` constituent in the right-hand side of `-np->det-cnp-`. `(? a 1s 2s 3s)` denotes a parser variable named `?a` whose possible values are `1s`, `2s`, and `3s`, meaning first-, second-, and third-person singular. In other words, this feature prevents the rule from matching plural common noun phrases, which are handled by another rule. Similarly, the `lex` feature of the `Determiner` constituent contains an *exclusive* parser variable (denoted by the `!` before the variable name `l`) that rules out determiners like “no” or “which”, which must be handled by different rules.

When a rule is applied to a sequence of constituents, all parser variables must have the same value. Thus, subject-verb agreement is handled by simply having the same parser variable match the `agr` features of the subject noun phrase and the verb phrase. If the same variable is assigned to incompatible values (e.g., `3p` for the subject and `3s` for the verb phrase, as is the case in the ungrammatical phrase “the ducks eats”), unification fails and the rule does not apply.

Once unification succeeds, the parser applies the variable bindings from the right-hand side of the rule to instantiate a constituent based on the left-hand side. Thus, `-np->det-cnp-` bound to “the duck” will produce a `np` constituent whose `var` and `agr` features match those of the child `cnp`. The `sem` field requires additional processing, discussed below, but the initial step is to substitute in the bindings of the parser variables. Thus, the semantics of the new `np` will

initially consist of something like `(thereExists duck4150 (isa duck4150 Duck) )`.⁷

The grammar currently consists of 467 rules. These cover phrase types such as noun phrases (“the duck”, “another fresh apple”), verb phrases (“hit the ball”, “ate the bread”), adjective phrases (“very green”), and measure phrases (“3 miles”). Because CNLU uses a context-free grammar, rules can only match contiguous sequences of constituents. This restriction makes it more difficult to handle certain linguistic phenomena, such verb particles (e.g., “put a hat *on*”), but it helps keep parsing tractable. The parsing algorithm is also written in such a way that (1) each sequence of constituents is only considered once, (2) sequences that cannot match the right-hand side of any rules are quickly ignored, and (3) full feature unification is only performed when it has a chance to succeed.

The parse continues until all input tokens have been added to the chart, every applicable grammar rule has been tried, and all of the resulting constituents (with probability greater than the cutoff threshold) have been added to the chart as well. The output is the set of parse trees that cover the entire input, along with their semantics. If no complete parse trees can be found, the parse is considered *fragmented* and the constituents that cover the largest spans are output instead, along with adjacent constituents to cover the remaining tokens.

While the CNLU grammar covers a broad range of linguistic constructs, it still has gaps that can be difficult to spot. INLD and GEDEM help alleviate this problem by finding and flagging places where the grammar is erroneous or insufficient.

⁷ Choice sets and discourse variable substitutions make it a little more complicated than this, but the general idea is that the algorithm builds the semantics for the parent constituent using the semantics of its child constituents, as bound to parser variables during feature-based unification.

2.2.3 Semantics

Semantic interpretations of a sentence are built from *semtranses* (“semantic translations”). Semtranses are semantic templates that map a root word to one of its meanings, represented by concepts in NextKB. For example, Figure 4 shows the semtranses for the noun “duck”.⁸ `(isa :NOUN Duck)` means that the word “duck” can refer to an instance of the collection `Duck`, a type of waterfowl, while `(isa :NOUN Duck-Meat)` points to the collection `Duck-Meat`, referring to the food. While a semantic template can contain an arbitrary conjunction of CycL expressions, noun and verb semtranses typically consist of a single `isa` statement that indicates which NextKB concept the word might refer to.

duck - Noun - FN_Animals - 
 (and (isa :NOUN Duck))

duck - Noun - FN_Food - 
 (and (isa :NOUN Duck-Meat))

Figure 4. CNLU noun semtranses for the word “duck.”

The placeholder `:NOUN` stands for the discourse variable assigned to the word “duck”. The relevant grammar rule will substitute in the discourse variable at the appropriate time, filling in the template and applying it to the specific entities discussed in the piece of text. This substitution takes place when a grammar rule applies that has a *grammatical function*, expressed as a Lisp keyword, as one of its features on its left-hand side. For example, the rule `-np->det-cnp-` in Figure 3 binds the grammatical function `:NOUN` to the discourse variable from the common noun phrase (`?varcnp`). The code responsible for updating the `sem` field takes the initial semantics for the new constituent (i.e., the `sem` field of the left-hand side of the rule, with

⁸ Semtranses are stored in the KB as facts using the `fnSemtrans` predicate, but they are displayed this way in the KB browser for legibility.

its parser variables filled in) and replaces all instances of `:NOUN` with the bound discourse variable. The same goes for any other grammatical functions the rule happens to bind.

Thus, as the grammar builds up parse trees by combining smaller constituents into larger ones, it also builds up the semantics of the sentence by (1) combining the semantics of the child constituents into the parent's `sem` field and (2) replacing grammatical functions with discourse variables to instantiate the semantic templates provided by the `semtranses`. In a complete parse tree, all grammatical functions in the root constituent's `sem` field should have been replaced with discourse variables. In the event that this does not happen, CNLU drops that interpretation because it is incomplete.

NextKB contains over 70,000 `semtranses` across 12 parts of speech.⁹ Most of them are derived semi-automatically from FrameNet (Fillmore, Wooters, & Baker, 2001), a linguistic ontology that maps how meaning can be conveyed by lexical units and their arguments. Beginning with a manual mapping from words to NextKB concepts (e.g., “duck” to `Duck-Meat`), a script was then used to apply information from FrameNet (in particular valence patterns, discussed below) and generate an initial set of `semtranses` for CNLU. These `semtranses` have since been maintained manually. New `semtranses` have been added as needed, and old ones have been corrected as errors in the NextKB mapping have been found. However, progress has been slow, incremental, and limited to bugs that are encountered during development. INLD and GEDEM should make it easier to identify missing or incorrect `semtranses` going forward while allowing Companions users to help, not just CNLU experts.

⁹ Some parts of speech, such as quantifiers and conjunctions, have lexicon entries but no `semtranses`. The grammar is responsible for matching these words in various grammatical constructs and assigning the appropriate semantics.

2.2.4 Valence Patterns

CNLU captures the semantics of verbs with Neo-Davidsonian (Parsons, 1990) event representations. Events are represented explicitly as entities, rather than implicitly as predicates, and their participants are linked to the event entity with binary predicates called *role relations*.

For example, the semantics of “ate” in the sentence “Bob ate the wedge” can be seen in Figure 5.

`(isa eat4200 EatingEvent)` says what type of event occurred, while `(consumedObject eat4200 wedge4237)` and `(performedBy eat4200 bob4152)` encode what was eaten and who ate it, respectively.

The figure shows a web interface for selecting a frame semantic for the word "ate". At the top, there is a "Clear" button and a title "FrameSemantics 'ate' (TokenFn Sentence-3856432374-4146 (SpanFn 1 2))". Below this, there are two radio buttons, each followed by a blue box containing a list of frame semantic triples and a grey box containing the function name "FN_Ingestion".

Radio Button	Frame Semantic Triples	Function
☐	<pre>(and (isa eat4200 HavingAMeal) (consumedObject eat4200 wedge4237) (performedBy eat4200 bob4152))</pre>	FN_Ingestion
☐	<pre>(and (isa eat4200 EatingEvent) (consumedObject eat4200 wedge4237) (performedBy eat4200 bob4152))</pre>	FN_Ingestion

Figure 5. Choice sets for the word “ate” in the sentence “Bob ate the wedge.”

Neo-Davidsonian event representations are useful because they do not specify the arguments of the verb in advance. For instance, the same basic representation can be used for “Eat” (an imperative command), “Bob ate”, “Bob ate the wedge”, “Bob ate the wedge quickly”, “The wedge was eaten”, and “The wedge was eaten by Bob”. The only differences between these sentences are what information about the event is provided, such as role relations (who did the eating, what was eaten) or manner (“quickly”). This is much more flexible than a fixed predicate like `(ate bob4152 wedge4237)`, which only fits two of these sentences.

CNLU uses *valence patterns* to keep track of which syntactic arguments a verb can take and how they map to role relations. Each valence pattern is a set of triples specifying one valid configuration of arguments. The triples map a grammatical function and phrase type to a role

relation. For example, Figure 6 shows part of one semtrans for “eat”. Each box in the table contains a valence pattern, consisting of one or more triples. The highlighted box contains the valence pattern that applies to “Bob ate the wedge”, where the noun phrase bound to the :SUBJECT fills the `performedBy` role and the noun phrase bound to the :OBJECT fills the `consumedObject` role. Other valence patterns cover sentences like “Bob ate” or “Bob ate with a spoon”. Role relations are filled in automatically when CNLU binds a grammatical function with a phrase of the appropriate type, resulting in a conjunction of expressions like the one seen in Figure 5.

Most of the valence patterns in CNLU were constructed automatically from FrameNet. Along with an ontology of frames (such as `Ingestion` or `Animals`), FrameNet contains an annotated corpus of how these frames appear in natural text. Each annotation consists of a lexical unit (a word or phrase), the frame it evokes, and the frame elements seen in that sentence. To construct valence patterns for CNLU, the frame elements are mapped to role relations, and their syntactic information is extracted from the annotation to yield a triple. Thus, each annotated example in FrameNet of the way a verb is used becomes a valence pattern that informs CNLU how the syntactic arguments of the verb map to semantics.

Valence patterns are one of the most important parts of constructing semantic interpretations with CNLU. They are the sole link between an event and its participants. If the valence pattern for a sentence is missing, CNLU will have no way to apply the grammatical functions bound by the grammar and will be forced to abandon this interpretation. This is useful behavior when it comes to filtering out bad interpretations (say, a verb sense that never takes an :OBJECT) but it means that all valid configurations of a verb’s arguments must appear among its valence patterns.

eat - Verb - FN_Ingestion - 
 (and (isa :ACTION EatingEvent))

:OBJECT NP consumedObject		
:SUBJECT NP consumedObject		
:OBLIQUE-OBJECT NP consumedObject		
:SUBJECT	NP	performedBy
:OBLIQUE-OBJECT	(PPFn with)	deviceUsed
:SUBJECT	NP	performedBy
:OBJECT	NP	consumedObject
:SUBJECT	NP	performedBy
:OBJECT	NP	consumedObject
:OBLIQUE-OBJECT	(PPFn near)	eventOccursAt

Figure 6. CNLU verb semtrans for the word “eat.”

All of this makes valence patterns a prime target for debugging. The process for extracting valence patterns from FrameNet annotations is error-prone, in part because the annotations are on wild text, with all its variety, in part because the grammatical conventions used by CNLU and FrameNet are not a perfect match, and in part because FrameNet annotations focus on declarative sentences and offer poor coverage of questions and commands. Erroneous valence patterns and gaps in their coverage are difficult to spot at a glance, and they are one of CNLU’s most technically challenging types of linguistic knowledge to correct.

In short, valence patterns are crucial for CNLU, but they are buggy, incomplete, and hard to maintain. At the same time, their meaning is intuitive and the way the parser uses them is well-defined. This makes them a natural fit for INLD and GEDEM, which handle the technical side of valence patterns automatically while delegating the linguistic intuition to the user.

2.2.5 Type Checking

Whenever CNLU tries to bind a grammatical function to a discourse variable, it performs *type checking* as a sanity check. It uses the possible types of the discourse variable, as determined by the `isa` statements it appears in elsewhere in the `sem` field, to decide if a proposed binding is valid. For example, if `:NOUN` appears as an argument for a predicate, a proposed `:NOUN` binding will be checked against the `argIsa` constraints of that argument (see Section 2.1.2) to determine if it can appear in that position. If the type check fails, the interpretation is dropped. The same occurs when a grammatical function is used to fill in a valence pattern. If a valence pattern binds `:SUBJECT` to a role relation that can only be filled by an agent, that binding will fail if the bound discourse variable cannot be an agent, as determined by its `isa` statements. When a proposed binding fails the type check for a role relation, that valence pattern will be dropped.

Aside from the `isa` statements found in the `sem` field, type checking relies on two types of knowledge: `genIs` statements and disjointness information in the KB. Type checking in CNLU takes a permissive approach, only ruling out a binding if *none* of the interpretations of the discourse variable can possibly meet the necessary type constraints. If a `:SUBJECT` needs to be an agent to satisfy a role relation, CNLU will only rule out the binding if all of the collections of the discourse variable are *disjoint* with the collection `Agent-Generic`, meaning they can have no overlapping instances.

Type checking helps rule out nonsensical interpretations of a sentence, but it can introduce some subtle bugs. Because interpretations are dropped with no indication to the user, even experienced developers can miss how they are ruled out, while the type checking itself may depend on `argIsa`, `genIs`, and disjointness knowledge that may not be obvious. INLD solves

this problem by making these dependencies explicit, allowing users to discover when erroneous type knowledge is blocking a desired interpretation.¹⁰

2.2.6 *Choice Sets*

To handle syntactic and semantic ambiguity, CNLU uses *choice sets* that capture the space of possible interpretations for part of a sentence. One choice set contains all of the top-level parse trees for the sentence, while other choice sets capture the meanings of individual words. Semantic choice sets are created when a word has more than one semtrans for the same part of speech, as is the case with “duck” as a noun. When a noun constituent for “duck” is created, its `sem` field is populated with a choice set that contains both semtranses for the word. These are passed up the parse tree as normal, and any grammatical function bindings will apply to all of the choices (e.g., `:NOUN` will be replaced with `duck4150` in the `isa` statements for both `Duck` and `Duck-Meat`).

Choices are dropped when a new binding fails type checking, and they multiply when more than one valence pattern matches a new binding. In this way, the `sem` field of a top-level constituent contains the full set of possible semantics that are consistent with that parse tree. As a final processing step, the choices from all the parse trees are aggregated and installed in the working memory of a FIRE reasoner, along with other information about the parse. The choices are connected to the semantics they entail using logical clauses installed in the WM, taking advantage of the fact that FIRE’s working memory is implemented as an LTRE (Everett & Forbus, 1996). Thus, when either the user or a reasoning process selects the choice (`isa`

¹⁰ It is worth noting that type checking is only a coarse filter on sensible interpretations. Language is flexible, and CNLU must be permissive to give downstream reasoning options to work with and allow metaphorical interpretations. Thus, even in a perfect world, this knowledge will never be sufficient to rule out all undesirable interpretations.

duck4150 Duck), the LTRE will update its state such that (isa duck4150 Duck) is believed to be true in the appropriate context, making it a part of the chosen interpretation for the sentence.

Choice sets are considered mutually exclusive and collectively exhaustive. That is, if “duck” means Duck, it cannot also mean Duck-Meat, and there are assumed to be no other meanings of the word. This assumption fails if CNLU is missing a word sense or rules one out erroneously, such as by type checking or a mistake in the grammar. A major part of INLD as applied to the CNLU system is determining when this is the case so the missing word sense can be corrected.

Disambiguation is a difficult task that depends on context. For example, the meaning of “Mary had a little lamb” changes depending on whether Mary is a sheep, a woman having dinner, or neither. Because this type of judgment relies on subtle world knowledge and knowledge of the task at hand, CNLU does not impose a one-size-fits-all solution. Instead, it lets each application implement whichever disambiguation strategy works best for it. The suite of disambiguation strategies that have been used with CNLU includes analogical word sense disambiguation (Barbella & Forbus, 2013), abduction based on handwritten rules (Tomai E. R., 2009), abduction based on learned cases (Crouse, 2021), an LLM-based classifier trained on FrameNet data (Ribeiro D. N., 2023), and a more general LLM-based classifier based on verbalized choices and zero-shot prompting (Zhao & Forbus, 2025).

INLD disambiguates sentences incrementally as the system asks the user questions. Automatic disambiguation techniques are outside the scope of INLD because they are either inscrutable and cannot easily be corrected (LLM-based techniques) or because the type of knowledge they use would be difficult for a user to correct (rules or cases). More generally,

INLD is concerned with debugging a system’s linguistic knowledge; its goal is to ensure that the right choices are available, not that the right choices are chosen.

2.3 Verbalize

While the knowledge used by hand-curated NL systems is explicit and can be debugged by experts, it can be difficult for non-expert users to understand. CNLU is no exception. Lexicon entries, grammar rules, semtranses, valence patterns, and type information all use specialized knowledge representation formats with their own conventions and a limited pool of developers with enough experience to maintain them. While the right setup can shelter the user from the worst of this representational complexity, there still remains the problem of conveying to the user what interpretations the system has found so that the user can render judgment. In particular, for semantic parsers like CNLU, the system must be able to express predicate calculus interpretations in terms a non-expert user can understand.

One naive approach would be to use predicate calculus itself, either as-is or with light editing for legibility. The concept names in NextKB are intended to be read by the developers, and they hew closely to the way those concepts are expressed in English.¹¹ The KB is full of concepts with intuitive names like `Cat`, `HumanAdult`, or `performedBy`. These might convey different interpretations to a non-expert user well enough for the purposes of INLD, especially if the names are modified to use normal spacing, capitalization, etc.

However, the naive approach runs into problems immediately. Text generated this way will be disfluent, with no grammar to ensure the output is well-formed. Role relations make up a significant part of CNLU’s interpretations, yet names like `performedBy`, `animateDoer`, and

¹¹ Contrast this with the unique IDs assigned to concepts by Wikidata (Vrandečić & Krötzsch, 2014).

`travels` all express different grammatical relations between the arguments, making it hard for a simple rule to handle all of the different conventions. Furthermore, this approach has no way to handle predicates with more than two arguments. Finally, concept names in NextKB can be ambiguous. By itself, the name `Duck` could refer to the waterfowl or the type of meat, just as the word can. Developers use comment strings and the surrounding ontology to determine what a concept name means. Stripping away this context and asking non-expert users to choose between `Duck` and `Duck-Meat` is infeasible.

To solve these problems, I use a system called *verbalize* to convert predicate calculus into English text. Verbalize is a template-based natural language generation (NLG) system designed for use with NextKB. It uses canned strings to express individuals and collections and templates to express predicates and functions. Thus, a CycL statement like `(isa Kimble-TheCat Cat)` would be expressed as “Kimble is a cat”, where “Kimble” and “cat” are the canned strings for the respective concepts and “_____ is a _____” is the template for `isa` statements. Verbalize walks down each statement recursively, verbalizing its arguments and using them to fill in the template. Pseudocode for this algorithm can be seen in Figure 7. Note that the algorithm can take a selection function to choose between multiple candidate verbalizations, and it falls back to a simple string-based verbalization method (e.g., `Kimble-TheCat` as “Kimble-TheCat”) when no verbalizations are available.

Verbalize uses language facts inherited from OpenCyc, namely `prettyString` facts for canned strings and `genFormat` facts for templates, but it has developed in its own direction to meet the needs of NextKB and FIRE. The system supports basic subject-verb agreement, verb tense, optional pronominalization, and fact aggregation, and it has hooks for both reasoning-based and code-based custom handlers to deal with corner cases that are difficult for the standard

```

// Given predicate calculus expression e, selection function S:
// Return a string conveying the meaning of e in natural language.
Function VERBALIZE(e, S):
  V = {} // Set of candidate verbalizations.
  If IS-ENTITY(e):
    // Retrieve strings for e.
    V = RETRIEVE-PRETTY-STRINGS(e)
  Else:
    // Retrieve templates for the predicate or function of e.
    p = PREDICATE-OR-FUNCTION-OF (e)
    G = RETRIEVE-GEN-FORMATS(p)
    // Verbalize each argument of the statement.
    For ai in ARGUMENTS-OF(e):
      vi = VERBALIZE(ai, S)
    // Apply each template to the verbalized arguments.
    For g in G:
      V = V ∪ {APPLY-TEMPLATE(g, v1, ..., vn)}
  // Finally, select among the options or return a fallback string for e.
  Return SELECT-OUTPUT(V, S) or FALLBACK-VERBALIZATION(e)

```

Figure 7. The verbalize algorithm.

top-down, lookup-based approach. Verbalization is also contextualized, so that different concepts can be verbalized in different ways depending on the application.

Although verbalize is an NLG system, it is a comparatively simple one. It takes individual statements and converts them into English by walking through their arguments, so it tends to produce simple clauses that map directly to the original predicate calculus. This is sufficient for literal statements that convey the meaning of CycL expressions, but it is insufficient to produce paragraphs of text or texts with more complex linguistic structures. Likewise, it is a poor fit for Neo-Davidsonian event representations, where all of the participants of the event are stored as separate facts, requiring several stilted sentences to express what can be said compactly in language.

Even with these limitations, verbalize is exactly what is needed for INLD. Instead of relying on the names of concepts directly, it uses `prettyStrings` and `genFormats`, both generating more fluent text and allowing each concept to be dealt with individually. For

example, role relations like `performedBy`, `animateDoer`, and `travels` can have `genFormats` that capture the relationship between their arguments, regardless of how the concepts are named. Questions in INLD are short, so the direct mapping from predicate calculus is not a problem. Nor are Neo-Davidsonian event representations, since role relations need to be split out and checked individually anyway.

Importantly, `verbalize` does *not* rely on the same body of linguistic knowledge as CNLU. If it did, errors might end up being invisible because the same knowledge would be used to parse a sentence and show the results of the parse to the user. For example, if there was a hypothetical incorrect mapping from the word “cheese” to the concept `Bread`, CNLU would interpret “cheese” as `Bread`, which it would express back to the user as “cheese”. But because `verbalize` uses `prettyStrings` and `genFormats` that are independent from CNLU’s lexicon and `semtranses`, the error is revealed when CNLU verbalizes `Bread` as “bread”.¹²

The cost of using `verbalize` is coverage. Just as CNLU needs sufficient lexicon entries and `semtranses` to cover the breadth of text it needs to parse, `verbalize` needs sufficient `prettyStrings` and `genFormats` to cover the breadth of concepts it needs to express in natural language. Fortunately, `NextKB` inherited extensive `prettyString` and `genFormat` coverage from `OpenCyc`, and more have been added as needed, in particular to cover the concepts the CNLU uses in its interpretations. While these facts are hand-curated, and errors are a potential pitfall, they are simple and useful enough to make a solid foundation for INLD. That is, `verbalize` allows CNLU interpretations to be expressed in English for the purpose of asking the user diagnostic questions.

¹² The situation is a little more complicated than this, and with the right setup, useful traction can be gained by generating text with the same linguistic knowledge used to parse. Chapter 5 explores this line of thought further. For the immediate purposes of INLD, though, using a separate body of knowledge is a boon.

One final consideration is ambiguity. `Verbalize` does not guarantee that the text it produces is unambiguous. If the `prettyStrings` for two concepts are similar enough, they might not be distinguishable by the output of `verbalize` alone. For example, the concept `Wedge-GolfClub` has a pretty string of “wedge”, which is ambiguous with “wedge shaped thing”, the pretty string of `Wedge`, the concept for a wedge shape.

Rather than go through `NextKB` and attempt to make all `prettyStrings` unique—a process that would lead to cumbersome, disfluent strings—I have the INLD system add clarifying information to each collection when it is presented to the user. Certain collections in the KB are marked as “clarifying”, meaning they represent intuitive conceptual distinctions and are suitable for use in this manner.¹³ When collections presented as choices to the user, the system walks up the `genls` hierarchy to find a clarifying collection that the choices do not have in common, then verbalizes that as well. For example, the `prettyStrings` for `Wedge-GolfClub` and `Wedge` can be clarified by adding information about their parent collections: “wedge (a golf club)” vs. “wedge shaped thing (a geometrical figure)”.

Aside from INLD, the main applications of `verbalize` are currently the `SocialBot` (Nakos & Forbus, 2025), which uses it to present the answers to user questions that were determined via reasoning, and an LLM-based disambiguation system (Zhao & Forbus, 2025), which uses it to verbalize choices as input to an LLM in the same way INLD uses it to ask the user questions. See Section 5.2 for discussion of another, more complicated NLG system that builds on CNLU’s linguistic knowledge rather than `prettyStrings` and `genFormats`.

¹³ Another innovation inherited from `OpenCyc`.

3. Types of Errors

The purpose of my work in this thesis is to discover and correct errors in the linguistic knowledge of hand-curated NL systems. Before discussing how to accomplish this, it will be useful to go over what errors one might expect to find. As with the rest of this thesis, my discussion here is grounded in the specific types of knowledge used by CNLU. However, similar types of errors will arise in any hand-curated NL system, in particular those that build up explicit semantic representations of text.

I distinguish between *errors* in a system’s linguistic knowledge and the *symptoms* which those errors give rise to. For example, a missing or overly restrictive grammar rule (an error) might cause a sentence’s parse to fragment (a symptom). The goal of INLD and GEDEM is to uncover errors by exposing symptoms to users in ways that they can notice. As another example, if “cheese” is erroneously mapped to `Bread`, the incorrect mapping would be the error, while the symptom would be that “bread” appears in the list of possible interpretations for the word.

Table 1. Taxonomy of types of errors found in CNLU.

ID	Error	Category	Symptoms
A1	Missing lexicon entry	Lexicon	Identified by parser
A2	Incorrect lexicon entry	Lexicon	Fragmented parse or incorrect parse tree
B1	Missing grammar rule	Grammar	Fragmented parse or incorrect parse tree
B2	Incorrect grammar rule	Grammar	Fragmented parse or incorrect parse tree
C1	Restrictive type constraint	Semantics	Type checking rules out correct interpretation
C2	Missing valence pattern	Semantics	Valence pattern checking rules out correct interp.
C3	Missing word sense	Semantics	Identified by parser or no correct interp. at all
C4	Permissive type constraint	Semantics	Bad interp. allowed or generates bad paraphrase
C5	Incorrect valence pattern	Semantics	Bad interp. allowed or generates bad paraphrase
C6	Incorrect semantics	Semantics	Correct interp. missing or generates bad paraphrase
D1	Missing inheritance info	Ontology	Type checking rules out correct interpretation
D2	Incorrect inheritance info	Ontology	Bad interp. allowed or generates bad implication
D3	Missing disjointness info	Ontology	Bad interp. allowed or generates bad implication
D4	Incorrect disjointness info	Ontology	Type checking rules out correct interpretation

Table 1 shows a taxonomy of error types and their symptoms for CNLU. CNLU's knowledge can be broken down into four parts: the lexicon, the grammar, the semantics, and the ontology. Each of these can have different types of errors, leading to different symptoms:

- 1) Lexicon entries include mappings from surface forms to root words, parts of speech, and grammatical features. Any of these could be erroneous, or the lexicon may be missing an entry it needs.
- 2) Grammar rules are complex and can break down in any number of ways, ranging from inappropriate feature constraints to bad interactions with other rules. The complexity of these errors makes them the hardest type to debug.
- 3) Semtranses and valence patterns can be incorrect or missing, causing CNLU to produce implausible interpretations for a sentence or fail to produce a correct one, respectively. The `argIsa` constraints (see Section 2.1.2) on the predicates they use can also be incorrect, leading to the right interpretation getting ruled out by type checking.
- 4) Other type checking issues can arise from problems with the ontology. In particular, the `genIs` hierarchy can be erroneous or incomplete, as can the KB's knowledge about which collections are disjoint. Any of these cases can cause a type check to pass when it should fail or fail when it should pass. Because type checking is only intended as a coarse filter, the latter case is more important than the former.

Errors fall into two broad categories: errors of *restriction* and errors of *permission*.¹⁴ *Errors of restriction* occur when the system's linguistic knowledge is too restrictive to produce the full range of interpretations it should. During language understanding, they manifest as gaps in the system's understanding. For example, in the sentence "Bob ate the wedge", the missing

¹⁴ These are analogous to Type I and Type II errors, but given that I will be talking about both understanding and generation, I opt for new terminology rather than muddying the old definitions.

sense of the word “wedge”—a sandwich—would be a symptom of a missing semtrans, an error of restriction that keeps CNLU from producing the desired interpretation of the sentence. These types of errors are the main focus of INLD. Errors of restriction are marked in **bold** in Table 1.

Errors of permission occur when the system’s linguistic knowledge is too permissive, allowing nonsensical interpretations to creep in. Because CNLU produces all possible interpretations of a sentence, subject to its linguistic knowledge and certain probability cutoffs, errors of permission are not a major problem during language understanding. They waste computational effort, they may distract from the right interpretation, and they may cause confusion if a user sees them, but ultimately, they are no different from any other interpretation that doesn’t fit the current context. For example, if the word “cheese” maps to the concept `Bread`, no harm is done so long as the right interpretation (`Cheese`) is also present. The disambiguation process simply picks `Cheese` instead of `Bread` and reasoning can proceed.

But problems arise when generation is involved. An NLG system using the erroneous mapping will generate “cheese” when it means `Bread`, causing problems and confusion. While Companions’ NLG needs have been modest so far, with verbalize’s template-based approach sufficing (Section 2.3), its long-term goal of fluent natural language interaction points to the need for a more robust, linguistically informed NLG system. I discuss such a system, called *say*, in Section 5.2 and show how it can reveal errors of permission using the GEDEM framework. Finally, errors of permission can point to inadequacies or shortcuts in an NLU system’s linguistic knowledge, making them worth finding and correcting even in the absence of generation. Errors of permission are marked in *italics* in Table 1.

Table 2 shows how the frameworks presented in this thesis map to the two kinds of errors and the inputs over which they operate. INLD focuses on errors of restriction and debugs

individual sentences in depth. Its goal is to find and debug specific errors in the system’s linguistic knowledge to aid language understanding. GEDEM uses a corpus of generated examples to find errors of permission at scale, but it does not drill down on any specific example.

Table 2. Debugging frameworks by input and category of error.

	Errors of Restriction	Errors of Permission
Single Sentence	INLD	<i>(Future work)</i>
Multiple Sentences	<i>(Future work)</i>	GEDEM

This division of responsibilities matches the current bottlenecks in Companions’ language capabilities. Understanding usually produces at least a reasonable partial interpretation, but it can be stymied by individual bugs and gaps in its coverage. INLD addresses these and lets errors be patched as they are discovered. Meanwhile, generation is still in its early stages, so it is sensitive to lingering errors in CNLU’s linguistic knowledge that are not obvious during understanding. GEDEM supplies a way to find these errors so they can be fixed.

The two remaining cells in the table correspond to sentence-level debugging of errors of permission and corpus-level mining for errors of restriction. Both of these pose additional challenges. Debugging errors of permission at the sentence level requires a reliable means of generating text to reveal those errors. While say can handle this in some cases, and it is a good fit for the problem in the long run, its coverage does not yet match that of CNLU. As say develops, having a framework for debugging errors of permission at the sentence level will become more important, but I leave this for future work.

The last cell in table, mining for errors of restriction at the corpus level, is currently limited by disambiguation. Error mining works by correlating words, phrases, or pieces of

linguistic knowledge with an error signal, such as whether the parse fragments. But finding an error signal for semantics is more difficult. GEDEM handles this by starting with an interpretation and generating sentences from it, then eliciting judgments about these sentences that can be correlated with the linguistic knowledge used to generate them. But to mine for errors of restriction, the system would first have to parse wild text, disambiguate it, and generate paraphrase from there. If the disambiguation system is flawed, it is difficult to tell whether to attribute a bad paraphrase to a linguistic error or the disambiguation system picking the wrong interpretation. As CNLU's disambiguation capabilities improve, this will be a promising area of future work.

Lastly, it is worth noting that the difficulty of these four quadrants depends on the nature and limitations of the NL system being debugged. Errors of permission are more difficult to find in systems without a generation component, while questions like how the system handles semantics, how it disambiguates sentences, and what types of introspection are available will influence how error mining and debugging take shape. CNLU has its own particular traits that have shaped its implementations of debugging and error mining. Other systems will fit these frameworks in different ways depending on their algorithms, use cases, and maturity.

4. Interactive Natural Language Debugging

Interactive Natural Language Debugging (INLD; Nakos, Kuthalam, & Forbus, 2022; Nakos & Forbus, 2024) is a framework for debugging hand-curated NL systems by asking the user a series of questions in natural language. The basic intuition is to separate the technical knowledge required to correct the system from the commonsense knowledge required to understand the meaning of a sentence. With the right setup, the former can be handled by the system itself, reducing the need for expert intervention, while the latter can be provided by a user with no specific expertise, greatly expanding the pool of people who can help debug the system.

The crux of this approach is to convert the workings of the NL system into a reasoning problem that can be used to diagnose and correct errors, with the user acting as a language-savvy oracle to allow the system to determine the intended meaning of the sentence. The fact that hand-curated NL systems use explicit linguistic knowledge means that there is a finite set of discrete facts for the system check, in contrast with ML-based systems that must be corrected with fine-tuning (Bommasani, et al., 2021), reinforcement learning (Bai, et al., 2022), or retraining. In essence, each new sentence is its own reasoning problem that can be used to check the system’s knowledge.

An example run of INLD can be seen in Figure 8. In the sentence “Bob ate the wedge”, “wedge” refers to a type of sandwich. However, CNLU only has noun semtranses for two senses of “wedge”: the shape and the kind of golf club. Thus, a word sense is missing, error type C3 in the taxonomy present in Chapter 3. INLD begins by parsing the sentence, then begins checking its interpretation against the user’s understanding. It starts by asking for the part of speech of “Bob”, allowing it to disambiguate the two parse trees it found, and then asks for the intended meaning of “wedge” in this context. When the user responds that neither is correct, the system

concludes that a word sense is missing. The missing sense can then be filled in by the user with a sequence of follow-up questions or flagged for an expert to handle later.

```

What part of speech is "bob"?
1) noun
2) proper noun

(Please enter a list of numbers between 1 and 2, or "none".)
> 2

What does "wedge" mean?
1) wedge (a golf club)
2) wedge shaped thing (a geometrical figure)

(Please enter a list of numbers between 1 and 2, or "none".)
> none

These assumptions are faulted:
-Choice Set #8748 ("wedge") is complete.
```

Figure 8. Example INLD transcript for the sentence “Bob ate the wedge.”

In this chapter, I describe the INLD framework, show how it applies to CNLU, and report experimental results demonstrating its utility for diagnosing and correcting errors. I begin with an overview of the four-stage INLD pipeline (Section 4.1), including considerations about error types and the limitations of this approach. Next, I explain how the first two stages of the pipeline can be formulated as a *model-based diagnosis* problem (Section 4.2.1) and walk through how a CNLU parse trace can be cast as such a problem (Section 4.2.2). I then discuss the debugging strategies for CNLU that build on this basic model (Section 4.3). I conclude by evaluating my INLD system for CNLU on sentences drawn from a causal reasoning corpus (Section 4.4).

4.1 The INLD Pipeline

INLD consists of a pipeline with four stages, as shown in Figure 9. The input is a sentence suspected to contain an error, and the output is a proposed correction to the system’s linguistic knowledge. Sentences with potential errors can be identified in several ways: from

failed regression tests, by the failure of downstream reasoning tasks, by manual inspection, or by systematic error mining.

The first stage of the INLD pipeline is *identification*. The input to this stage is the suspect sentence, and the output is a set of *symptoms* that characterize a potential error, as discussed in Chapter 3. Some symptoms, such as fragmented parses or unknown words, can be detected automatically, reducing this stage of the pipeline to a series of automatic checks. Other symptoms, such as missing word senses or incorrect parse trees, require a human to judge. The system will continue to ask the user questions until it has identified a symptom, or it has checked every possibility it can.

Symptoms guide debugging and serve as its success criteria. If a proposed correction fixes the symptoms without introducing new ones, debugging succeeds. Otherwise, the system may need to backtrack or initiate another round of debugging.

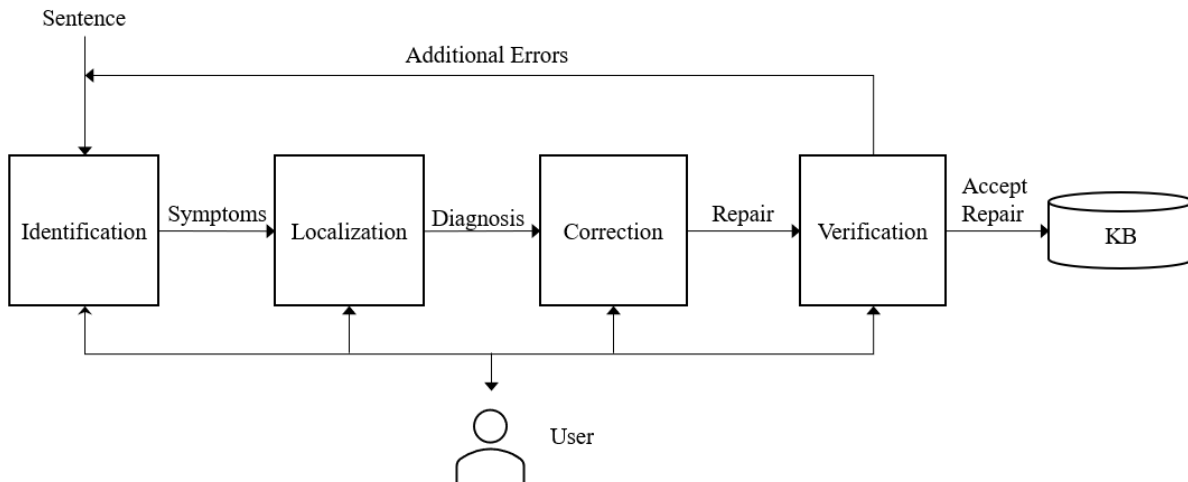


Figure 9. The INLD pipeline.

The second stage of the INLD pipeline is *localization*. This stage takes a set of symptoms and attempts to locate the error that caused them, whether that is a piece of knowledge that is incorrect or one that is missing. Like symptom identification, the complexity of localization varies with the type of symptom and the type of error. In the case of an unknown

word (A1 in Table 1), for example, localization is trivial based on the symptom: a lexicon entry is missing, and likely the semtrans to go with it. In the case of a fragmented parse, though, the situation gets more complex. Determining that a needed grammar rule is missing (B1) might require extensive drill-down and checking as the system sifts through a huge space of possibilities. This type of reasoning is non-trivial and requires both introspection by the system and targeted diagnostic questions to the user. I discuss this problem further in Sections 4.2 and 4.3.

The output of the localization stage is a *diagnosis*, a specification of what knowledge that is incorrect or missing that would explain the observed symptoms. The diagnosis is the most important product of INLD, as it contains enough information for an expert to fix the error with a minimum of effort, even if the system is unable to correct the error itself. As long as a non-expert user can complete the diagnosis stage with the help of the system, INLD has achieved its primary goal of expanding the pool of users who can help debug the system.

The third stage of the INLD pipeline is *correction*. Given an error diagnosis produced by the previous stage, the system works with the user to repair the error if possible, asking more questions to fill in the gaps in its knowledge. Like the previous two stages, the complexity of correction depends on what error needs to be corrected. Some errors are trivial to correct given the diagnosis, such as when an incorrect word sense needs to be removed (C6). Others require additional information from the user, such as the grammar features of a new lexicon entry (A1) or the collection that a new noun semtrans should point to (C3). Still others, such as a missing grammar rule (B1), are too complicated to realistically debug by asking a non-expert user questions in natural language. If correction is possible, the output is a *repair*, a proposed change to the system's linguistic knowledge.

The fourth and final stage of the INLD pipeline is *verification*. Once a repair has been proposed, verification is responsible for checking its validity before it is added to the KB. The extent of verification varies depending on the resources available. *Local* verification only checks the repair against the current sentence, making sure that it resolves the observed symptoms. If the symptoms remain, or if new symptoms appear, the system can perform another round of INLD to attempt to determine why.

Global verification checks the repair against other sentences to see if it has any undesired side effects. In an ideal world, all of the system’s knowledge would be verified against a growing body of test examples. No change would be made until its full ramifications were known, and thousands of saved examples would let the system determine semi-automatically when something broke. In practice, global verification requires extensive infrastructure to support, along with methods for resolving contradictions between examples and heuristics for managing the limited attention users have for answering questions.

Notably, INLD is not an all-or-nothing proposition. Even partial information about an error can help an expert get started on debugging it manually, so the process is worthwhile even if a complete repair cannot be found. Expanding the pool of users who can debug the system, even if that debugging is incomplete, frees experts up to focus on more challenging problems, such as dealing with new linguistic phenomena or improving the NL system’s algorithms. The same goes for error types: An INLD pipeline does not need perfect coverage of the kinds of errors found in the underlying NL system in order to be useful. In fact, the kinds of errors that are easiest to check for and correct—missing coverage for lexicon entries and semantics—are the ones where an expanded user pool would help the most.

My CNLU implementation of INLD focuses on diagnosis, discussed in the next section. While correction and verification are important for completing the INLD pipeline and allowing non-expert users to improve the system’s linguistic knowledge, they pose distinct challenges. Every error type requires its own correction method, and even something as straightforward as adding a new noun semtrans requires searching NextKB for an appropriate collection. Given the breadth of the ontology, the subtleties of proper ontologization, and the limits of an NL interface for this type of task—not to mention similar issues with other error types—I leave correction (and thus verification) for future work.

4.2 Interactive Error Diagnosis

The core part of the INLD pipeline is its first two stages: symptom identification and error localization. In the worst case, diagnosing the error constitutes an elaborate, multi-step reasoning problem that requires a detailed model of the underlying NL system. While there are several approaches that could work, such as constructing a constraint satisfaction problem or using a fixed “flowchart” of questions, I have chosen to implement the first two stages of the pipeline using a *model-based diagnosis* framework (de Kleer & Williams, 1987), previously reported in Nakos & Forbus (2024).

Model-based diagnosis has several desirable properties for INLD. First, it is a well-studied problem with existing algorithms and a range of successful applications (Isermann, 2005). Second, by its nature, it is a natural fit for error diagnosis. Third, model-based diagnosis algorithms are not limited to finding a single error at a time, so INLD will not fail if a sentence has multiple errors. Finally, model-based diagnosis is a flexible framework that can

accommodate the kinds of corner cases that occur when diagnosing hand-curated NL systems, such as locating where a desirable word sense is incorrectly ruled out.

Model-based diagnosis compares the behavior of an *artifact* to the predictions of a *model*. It uses the discrepancies to determine where the artifact is malfunctioning (taking the model as ground truth) or where the model fails to capture its behavior (taking the artifact as ground truth). The artifact can be physical, like an aircraft outflow valve or the steering behavior of a car (Isermann, 2005), or conceptual, like a student's reasoning on a homework problem (de Koning, Bredeweg, Breuker, & Wielinga, 2000).

In the case of INLD, the artifact is the user's understanding of the sentence, and the model is a trace of the parser's behavior as it tries to interpret the same sentence. Because the user understands language in a way the parser is trying to emulate, the user's understanding is taken as ground truth. Thus, any incompatibilities between the user's understanding and the parser's interpretation indicate a *fault* in the model, corresponding to a gap or error in the parser's knowledge.

Naturally, the user's understanding of a sentence cannot be examined directly, so the system must ask the user questions to capture relevant aspects of it. Ideally, these questions would correspond directly to elements of the parse trace, allowing the algorithm to check every step of the parse, but in practice this is infeasible. The parser relies on types of formal knowledge that a non-expert user will not understand, such as grammar rules, many of which cannot be easily expressed as questions. Thus, the system can only ask questions that (1) a non-expert user is likely to be able to answer, (2) can be expressed easily in natural language, and (3) shed some light on the desired interpretation of the sentence. It is up to the designer of the INLD system to choose questions that provide enough leverage for reasoning while still being

accessible to the user. Examples include questions about word senses, parts of speech, or basic aspects of the sentence structure.

Asking the user a question is analogous to taking a *measurement* to diagnose a fault in a physical system. The same way that inspecting the output of various parts in a printer can be used to triangulate which one is broken, targeted questions can be used to determine where the user's understanding of a sentence differs from the interpretation constructed by the NL system. However, just as not all parts of a machine are easy to inspect, not all elements of a parse can be directly expressed as questions. Those that can are referred to as *measurement points*, constituting the pool of potential questions to ask.

In the remainder of this section, I describe an implemented diagnosis engine for CNLU. The engine takes a sentence as input, parses it with CNLU, converts the parse trace into a model-based diagnosis problem, and asks the user questions until the algorithm finds a diagnosis which explains any discrepancies between CNLU's behavior and the user's understanding of the sentence. This diagnosis specifies which pieces of linguistic knowledge are erroneous or missing. In some cases, these can be corrected with further help from the user.

While the distinction between symptom identification and error localization is important for INLD, in a model-based diagnosis framework, both are handled similarly. The variable nature of both stages means that, depending on the symptom and the error, either or both might not require asking any questions (e.g., automatically detecting that the lexicon has no entry for a word). As a general pattern, though, the algorithm will first find a fault representing a symptom (e.g., a missing word sense in the interpretation), then drill down further to find a second fault representing the underlying error (e.g., a missing semtrans).

4.2.1 GDE & CATMS

The backbone of the diagnosis system for CNLU is the General Diagnostic Engine (GDE; de Kleer & Williams, 1987). GDE is a model-based diagnosis algorithm that eschews domain-specific diagnosis strategies in favor of a general approach that depends only on the model itself and the set of possible measurements. Given a model and the *observations* made so far, GDE computes which measurement to take next that will best distinguish among the current set of *minimal diagnoses*, minimal sets of faults that suffice to explain the observed discrepancies between the artifact and the model. The algorithm terminates when only one minimal diagnosis remains, in which case that diagnosis is output, or when no more measurements are possible, in which case either there are no discrepancies or they cannot be explained with the set of measurements available to the model.

GDE uses an assumption-based truth maintenance system (ATMS; de Kleer, 1986) to keep track of its hypotheses, allowing the set of minimal diagnoses to be computed efficiently in parallel. The purpose of an ATMS is to track the truth value of a set of assertions across all possible logical *environments*. Each environment consists of a set of *assumptions* believed to be true. Inconsistent environments are marked as *nogood* and indicate that the assumptions are contradictory. Environments are inconsistent if they entail an assertion marked as a *contradiction*. The ATMS maintains a set of *nodes*, each of which represents an assertion, assumption, or contradiction. Every node has a *label* that compactly represents the set of environments in which the node holds true. For a contradiction, its label is a set of environments to mark as *nogood*.

Environments are represented as sets of assumptions: $\{A, B\}$ is the environment where assumptions A and B (and only those assumptions) are true. One environment *subsumes* another

if the first environment only contains a subset of the second one's assumptions. Thus, the environment $\{A, B\}$ subsumes the environment $\{A, B, C\}$. Environments are monotonic, in that new assumptions cannot cancel an old one. All assertions entailed by an environment are true in the environments it subsumes. If an environment is nogood, all environments it subsumes will be nogood as well.

The label of a node is the minimal set of environments in which it is true. For example, the label $\{\{A, B\}, \{C\}\}$ indicates that the corresponding assertion holds in environments where A and B are true or where C is true. There is no need to store larger environments such as $\{A, B, C\}$ because they are subsumed by the environments already in the label.

The ATMS represents the dependencies between assertions using *justifications*, which link a set of antecedent assertions to a consequent. For example, the justification "A & B $\rightarrow$ D" means that assertion D is true in all environments where A and B are true. The algorithms that keep the ATMS up to date propagate labels from antecedents to consequents, taking the cross product of the antecedent labels, adding them to the label of the consequent, and then removing all non-minimal environments.

The price of keeping track of all environments at once is the risk of an exponential blow-up in the size of labels. The greater the number of assertions in the ATMS, and the more complicated the justification structure, the larger the labels of its assertions. Eventually, the algorithm will become intractable. This is the case with INLD: The number of assumptions needed multiplies rapidly with the number of possible interpretations, and the complexity of the justification structure compounds the problem. With a naive ATMS implementation, even short sentences can become intractable.

To keep the size of the labels in check, I use a CATMS (DeCoste & Collins, 1991), an ATMS variant that *compresses* its labels to prevent unnecessary blowup. Any node marked as an assumption will only have its own singleton environment as a label, no matter what other environments entail it. Thus, if A is an assumption, its label will always be $\{\{A\}\}$, even if it is justified by other assertions.

To support this, the CATMS replaces the standard environment subsumption relationship with *c-subsumption*, a more complicated variant that takes the assumptions implied by an environment into account. For example, the environment $\{A\}$ c-subsumes any environment that implies A. Thus, the environments that would in the label of A in the ATMS can be compressed to just $\{\{A\}\}$ without losing any information, at the cost of a more complicated subsumption check at runtime. With the right caching of these environments, the check for c-subsumption can be made efficient, minimizing the runtime cost of using a CATMS while reaping the benefits of much smaller label sizes.

In practical terms, this means we can prevent the combinatorial blowup of labels by turning select nodes into assumptions. Whenever the labels grow unwieldy, we pick an intermediate assertion and convert it into an assumption to curtail the label size. This works especially well when the justification structure has bottlenecks that can be converted into assumptions. In essence, a CATMS allows one large reasoning problem to be partitioned into smaller ones, maintaining the environmental completeness of the ATMS without letting the labels get out of hand. This is a natural fit for INLD, where nodes separate naturally into clusters whose behavior can be summarized with a single output assumption. The efficiency gains from a CATMS are crucial to allow the system to run in real time.

For the GDE algorithm, the most important capability of a (C)ATMS is the *interpretations* function. The input to this function is a set of *choice sets*¹⁵, each of which contains a set of assumptions to select from. The function takes all combinations of choices, one from each choice set, and returns the subset of those environments that are consistent. In addition, the function takes a set of *defaults*, assumptions that should be assumed true if possible. For each environment constructed from the choice sets, the function creates additional environments containing each maximal subset of the defaults that are consistent with the rest of the environment. For example, suppose the *interpretations* function is given the choice sets (A, B), and (C) and the defaults D and E. If {A, D} is nogood and {D, E} is nogood, the *interpretations* function will output {A, C, E}, {B, C, D}, and {B, C, E}.

GDE uses this function to construct the set of minimal diagnoses for a model. It takes a list of assumptions corresponding to each component of the model behaving correctly and passes it in as the list of defaults. The *interpretations* algorithm then returns all maximal sets of defaults that are consistent, each of which constitutes a *diagnosis*. Any defaults that are not present in a diagnosis environment are considered *faulted*. The algorithm then takes the subset of these diagnoses that have the minimum number of faults (i.e., where the most components are working) and determines the next measurement to take to best choose between them.

The way this works can best be seen with a simple example. Suppose the model contains three components, X, Y, and Z, along with other assertions and justifications reflecting the logical dependencies between them and the observations made so far. GDE calls the *interpretations* function to find the current set of minimal diagnoses. The list of defaults is the

¹⁵ These choice sets are unrelated to the choice sets produced by CNLU, as are the “interpretations” produced by the function. To avoid confusion, I will only discuss ATMS choice sets and interpretations in these paragraphs.

assumptions “X is okay”, “Y is okay”, and “Z is okay”, abbreviated $ok(X)$, $ok(Y)$, and $ok(Z)$. If the function returns the environment $\{ok(X), ok(Y), ok(Z)\}$, all of the components can be okay at the same time and there is not yet any evidence for a fault. If the function returns $\{ok(X), ok(Y)\}$ and $\{ok(X), ok(Z)\}$, then there is no way for all of the components to be okay at the same time, so at least one of them must be faulted. The minimal diagnoses here are either that Z is faulted (thus $ok(Z)$ is false) or Y is faulted (thus $ok(Y)$ is false). The GDE algorithm will then pick a measurement to take that will help determine whether Z or Y is faulted. Specifically, it will choose the measurement whose outcomes will provide the most information about the current set of minimal diagnoses.

The vanilla GDE algorithm does *not* pass any choice sets into the *interpretations* function. It only cares about the defaults (i.e., okayness assumptions for the set of faultable components) so it can determine a diagnosis. For INLD, I modify this arrangement slightly. The choice sets are pairs of *completeness assumptions* (discussed below), which may be faulted, and singleton choice sets used to store information about problem state. The pairs reformulate GDE faults so that both okayness and not-okayness are represented explicitly. This proved to be more practical for capturing the logic of a CNLU parse; faults occur when the “okay” member of a pair is excluded from the minimal diagnosis and its corresponding “not okay” assumption is included. And due to the way *interpretations* constructs its environments, it is far more efficient to pass these pairs in as choice sets rather than defaults.

Meanwhile, the singleton choice sets allow INLD to store information about the problem state and back out of guesses gracefully. Because an ATMS is monotonic, deletions are difficult. To avoid making any irreversible changes to the ATMS state, all observations (answers provided by the user) and any problem state assumptions (for decomposition, detailed in Section 4.3) are

passed to the *interpretations* algorithm as singleton choice sets, rather than being asserted as true in the ATMS. Thus, every environment output by the function will contain the assumptions corresponding to all observations and all current problem state assumptions, but they can just as easily be dropped from future calls, allowing backtracking if needed.

4.2.2 Problem Formulation

Using GDE and a CATMS as building blocks, I next discuss how to cast interactive error diagnosis, the identification and localization stages of INLD, as a model-based diagnosis problem, beginning with the diagnosis algorithm illustrated in Figure 10. The algorithm consists of an *inner loop*, corresponding to the GDE algorithm, and an *outer loop* that examines the diagnosis produced by GDE and *decomposes* the model if necessary (akin to the decomposition process used by de Koning et al., 2000). The inner loop asks the user questions repeatedly, taking measurements until a minimal set of faults has been found. The outer loop is responsible for drilling down if necessary, taking the initial diagnosis and reformulating the model so that GDE can produce a more specific diagnosis next time around. The corresponding pseudocode can be seen in Figure 11. (Note that the decomposition method depends on the fault type.)

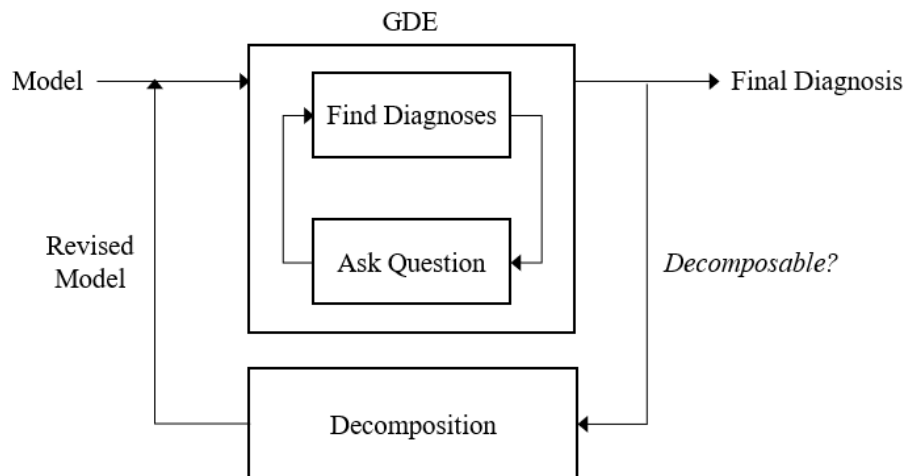


Figure 10. INLD error diagnosis loop.

```

// Given sentence s:
// Return a diagnosis of any errors in CNLU's parse of s.
Function INLD(s):
  PARSE(s, {}) // Parse the sentence with no constraints.
  M = INSTALL-PARSE-IN-ATMS(s) // Initial model of the parse.
  O = {} // Set of observations (user's answers).
  // Get the set of minimal diagnoses for M consistent with O.
  D = MINIMAL-DIAGNOSES(M, O)
  // Outer decomposition loop.
  While TRUE:
    // Inner GDE loop.
    While |D| > 1 and MEASUREMENT-AVAILABLE(M, O):
      // Find the measurement to take that best distinguishes D.
      m = BEST-MEASUREMENT(M, O, D)
      // Take the measurement and record the user's answers.
      O = O  $\cup$  ASK-USER(m)
      // Recalculate the minimal diagnoses.
      D = MINIMAL-DIAGNOSES(M, O)
    // Decompose one of the diagnosed faults, if possible.
    If |D| == 1 and HAS-DECOMPOSABLE-FAULT(D0):
      f = BEST-FAULT-TO-DECOMPOSE(D0)
      // Reformulate the model by decomposing the chosen fault.
      M = DECOMPOSE-FAULT(M, f)
    Else:
      Return D

```

Figure 11. INLD model-based diagnosis algorithm.

For example, the first round of GDE might yield an initial diagnosis that a word sense is missing from the choices produced by CNLU, a symptom that could correspond to several underlying errors depending on why the word sense is missing. The outer loop would then decompose the model further, installing new assertions, assumptions, and justifications in the ATMS to capture these possibilities. For instance, a word sense might be missing because it was ruled out by type checking, because none of the constituents it appeared in reached a top-level parse tree, or because there was no semtrans for it in the first place. Knowing that the desired sense for a particular word is missing, the outer loop walks through the parse trace to find other senses that were ruled out or dropped, then installs them in the ATMS. The next round of GDE will ask the user if any of these failed word senses was the desired one. If so, the problem can be

decomposed further to find the error that caused the desired word sense to fail. If none of the failed word senses are right, the algorithm can conclude that a semtrans is missing. In either case, the cycle continues until the algorithm finds a diagnosis that cannot be decomposed further, corresponding to the underlying gaps or errors in CNLU's knowledge that caused the symptoms.

There are several reasons to use an algorithm with an inner and an outer loop, as opposed to cramming everything into one round of GDE. The first is *tractability*. While the CATMS helps keep the runtime of the algorithm manageable by keeping labels small, the overall size of the problem is still a significant concern. The amount of information produced by CNLU is quite large. Representing the top parse trees in FIRE's working memory frequently takes hundreds or thousands of facts, and the full parse trace may have thousands of constituents, each with its own grammatical features and semantics. While all of these elements are part of the parse and should be accessible to INLD, the vast majority of them are not relevant. By allowing decomposition, the algorithm can include only a subset of this information in its model in the first pass, then install other relevant subsets as needed. For example, there is no need to install dropped word senses in the ATMS until there is evidence that a desired interpretation might have been ruled out along the way. Decomposition allows the algorithm to only deal with the logical structure that it needs to diagnose the error.

The second reason decomposition is useful is for *control*. The *decomposition strategies* that drive the outer loop provide fine-grained control of how the model is explored. When more than one decomposition is possible, the outer loop picks one and installs a *problem state assumption* that the inner GDE loop will treat as true. If the assumption doesn't pan out and GDE hits a dead end without finding the underlying error, the outer loop can retract the assumption (by removing it from the choice sets passed to the *interpretations* function) and move

on to the next possible decomposition. Thus, decomposition strategies provide macro-level control flow for debugging, while the GDE algorithm selects measurements to diagnose the faults in a given problem state.

The final reason for the structure of the algorithm is *modularity*. Decomposition strategies are a natural way to break the diagnosis down into manageable pieces. Extending an INLD system to handle a new type of error becomes a matter of finding the right decomposition strategies and creating the right handlers for them, so the algorithm can install more problem state and ask the user the appropriate questions. New strategies can be added to the library as they are formulated, allowing the INLD system to improve its coverage incrementally. INLD is not an all-or-nothing proposition, and nowhere is that more evident than in decomposition.

For CNLU's INLD system, the outer loop is implemented using *solve*, an incremental problem-solving engine for FIRE (Paritosh & Forbus, 2005). Solve builds an AND-OR graph that decomposes *goals* into subgoals according to *suggestions* for how to solve them. The algorithm maintains an agenda of nodes that should be worked on. Beginning with the lowest-cost goal, the algorithm attempts to either solve that goal by querying FIRE or decompose it with an appropriate suggestion. When one subgoal is solved, its variable bindings are used to instantiate the next subgoal in its parent suggestion, until a complete solution for the parent goal has been found.

Solve suggestions are a natural fit for the outer loop of INLD. They are flexible, they can solve the same goal in multiple ways, and one suggestion can be prioritized over another by assigning it a lower cost, allowing easier decomposition strategies to be explored before harder ones. An example solve suggestion from CNLU's INLD system can be seen in Figure 12. It is responsible for determining what measurement to take next, invoking the GDE inner loop.

The name of the suggestion is boxed in red. The goal pattern that the suggestion applies to is boxed in green. The subgoals are boxed in orange. The first two subgoals are outsourced predicates in FIRE; they run Lisp code to query the underlying ATMS problem state. In this case, they compute the minimal diagnoses that might explain the current problem state (invoking *interpretations*) and the measurement (question for the user) that should best distinguish between them. The last subgoal has its own set of solve suggestions, not shown. These handle three different cases: when INLD has found a single diagnosis and can attempt decomposition, when INLD has a measurement to take to help distinguish between its remaining diagnoses, and when INLD has run out of measurements and should terminate. The AND-OR graph expands hierarchically until INLD runs out of measurements or a single diagnosis is found.

```
(defSuggestion DiagnoseINLD
  (inldDiagnose ?inld ?state)
  :test (inldFeatures ?inld :diagnose)
  :subgoals ((outsourcedOnly (inldMinimalDiagnoses ?inld ?state ?diagnoses))
             (outsourcedOnly (inldBestMeasurement ?inld ?state ?diagnoses ?m))
             (inldMaybeMeasure ?inld ?state ?diagnoses ?m))
  :documentation "Perform one diagnosis (identification and/or localization)
                  step for INLD. This will be called recursively until no
                  more measurements need to be taken.")
```

Figure 12. Solve suggestion used for INLD diagnosis.

With the diagnosis algorithm in place, I next turn to the model it will operate over. CNLU traces contain a significant amount of information, so knowing what to extract for diagnosis is crucial. For the initial version of model, the goal is to capture basic information about the interpretations of the sentence produced by CNLU, including the content of the semantic choices and some basic information to distinguish between parse trees (namely, parts of speech for words in the sentence). Decomposition can then consult the parse trace again and install more structure as it is needed.

For CNLU, the model construction process breaks the parse trace down into a set of *parse elements*. Parse elements include parse trees, semantic expressions, constituents, choices,

or any other meaningful piece of data constructed during parsing. They may also include any intermediary constructs that are helpful for debugging, such as simplified parse trees or variable bindings. To avoid bloat, parse elements are only added to the model when they are needed.

The crux of model formulation is deciding what faults are possible. For the initial model, faults represent symptoms of the underlying error. When the model is decomposed to explore a symptom further, the decomposition strategy can install new faults corresponding to either more specific symptoms or the error itself. For CNLU, faults are represented with *completeness assumptions*. A completeness assumption is a type of closed-world assumption that states that some set of parse elements is *complete*, meaning that one of the elements belongs in a valid interpretation of the sentence. For example, the choice set for the word “wedge” is complete if one of the choices captures the intended semantics of the word in the context of the sentence. The choice set would be *incomplete* if the right sense of “wedge” was missing. There can also be completeness assumptions for parse trees, semtranses for a word, or valence patterns for a semtrans: any spot where a parse element needed for a valid interpretation might be missing.¹⁶

While completeness assumptions are useful for linking faults in the model to symptoms and errors, they do a poor job of driving diagnosis. GDE needs to prove that an assumption is contradictory with the rest of an environment in order for it to be considered faulted. However, completeness assumptions are weak—in essence, “one of the elements in this set is part of a valid interpretation of the sentence”. By themselves, they do not cause the conflicts that the GDE algorithm needs to determine that they are faulted.

¹⁶ Completeness assumptions are the most useful when dealing with errors of restriction, where CNLU fails to produce the desired interpretation. Errors of permission are harder to debug at the level of individual sentences, since even a nonsensical interpretation of a word might make sense in another context. For example, the word “had” in “Mary had a little lamb” is generally taken to mean “owned”, but that is not grounds to say the “gave birth” sense of the word is erroneous, as there are other sentences where it makes sense. I focus on errors of restriction here and deal with errors of permission at a more coarse-grained level in Chapter 5.

Therefore, each completeness assumption is paired with a corresponding *incompleteness assumption*, which states that a given set of parse elements is incomplete. This is a much stronger claim—“none of the elements in this set belong in a valid interpretation”—which gives the GDE algorithm something to work with. In the ATMS, each pair of completeness and incompleteness assumptions is used to justify a contradiction, making any environment that contains both of them nogood. Then both assumptions are added to as a pair to the list of choice sets, so that GDE will be forced to take measurements to choose between them.

All else being equal, the algorithm will produce two sets of minimal diagnoses: one set of diagnoses where the completeness assumption is believed (and thus not faulted), and one set of diagnoses where the incompleteness assumption is believed (and thus its partner is faulted). As the corresponding environments will have the same cardinality, the GDE algorithm will try to find a measurement whose outcomes will choose between them. In practice, this means asking questions about the set of parse elements until either one is found to be valid (e.g., the correct sense of the word is present) or all of them are ruled out (e.g., there is no correct word sense). With one pair of completeness assumptions for each set of parse elements that can have an element missing, this setup allows the GDE algorithm to ask questions until it determines what, if anything, is missing from CNLU’s interpretation of the sentence.

As a side note, although incompleteness assumptions are part of the choice sets used by the GDE algorithm, it is not meaningful for one to be faulted. (That is, it is good when a set of parse elements is complete; the situation bears no further scrutiny.) As such, incompleteness assumptions are not considered faults and cannot trigger decomposition when they are false.

To put completeness assumptions to work, it is necessary to define what it means for a parse element to be “right”. This is complicated by two issues. First, sentences may legitimately

be ambiguous, in which case there is no single “right” interpretation to find. To use a familiar example, “Mary had a little lamb” has several meanings, multiple of which could be correct in a particular context. It would not do to adopt a definition of “right” that forces a sentence to have exactly one correct interpretation.

Second, a piece of knowledge may be correct even if it does not apply to the current sentence in the current context. There is no reason to treat the “giving birth” sense of the word “had” as erroneous just because the user does not think it applies. Any definition of “right” must take this into account, gracefully handling the common situation where a rule or word sense is perfectly fine but not currently applicable.

Thus, to accommodate ambiguity and context dependence, a parse element is considered *acceptable* if it belongs to a desired interpretation of the sentence and *unacceptable* if it does not.¹⁷ The system can reason about the parse by converting the logical relationships between the acceptability of parse elements into justifications in the ATMS.

For example, the first semantic choice for the word “ate” in Figure 5 is only acceptable if all of its conjuncts are acceptable. If “ate” does not refer to the concept `HavingAMeal`, if the consumed object is not the wedge, or if the action is not performed by Bob, the choice is unacceptable. Likewise, a choice set is complete (i.e., its completeness assumption holds) if at least one of its choices is acceptable, and it is incomplete if all of its choices are unacceptable. This basic setup—smaller parse elements supporting larger ones, and larger ones feeding into completeness assumptions—allows the logic of a parse to be cashed out as ATMS assertions, assumptions, justifications, and defaults.

¹⁷ “Desired” gives the user latitude to debug specific interpretations. Even if one interpretation is valid, it may not be the one the user wishes to focus on right now.

Decomposing parse elements this way has the advantage of factoring out common structure. For example, the two choices for “ate” in Figure 5 share the same role relations and differ only in what collection they use to represent the event. Thus, the system can gather information about both choices by asking about their shared expressions. If, somehow, Bob did not do the eating in this sentence, both choices would be deemed unacceptable, and the choice set would be flagged as incomplete.

Another advantage of using acceptability as a common currency is that it provides a clean way to formalize measurements. When the system asks the user a question—whether yes-or-no (e.g., “Is ‘wedge’ a noun in this sentence?”) or multiple choice (e.g., “What does ‘wedge’ mean?”)—it will cash out the user’s response as one or more *acceptability judgments* that reflect whether the user finds the corresponding parse elements acceptable. With enough such judgments, the algorithm can determine which completeness assumptions are faulted, identify symptoms, and localize the error.

When GDE has to distinguish between a set of minimal diagnoses, it looks at the set of measurements that have yet to be taken. For the initial model formulation, this is limited to asking about atomic semantic expressions and parts of speech, but decomposition strategies can install new measurement points as needed. The algorithm will then aggregate these measurements as much as possible, grouping yes-or-no questions about parts of speech, collections, or role relations into multiple-choice questions whenever possible. This lets the user provide more information with a single answer, speeding up debugging. Then the best question is chosen by the algorithm, it is converted into English with *verbalize*, and it is presented to the user.

Measurements are recorded in the algorithm's problem state. They are always treated as true for the purpose of calculating diagnoses. That is, no diagnosis may include an assumption that is inconsistent with the measurements taken so far. For example, if all senses of a word are unacceptable, its completeness assumption will be excluded from the diagnosis, making it faulted and signaling that the senses are incomplete.

Not all parse elements can be measured directly. Some are too complicated or specialized to express to the user. For example, no one but the most experienced users will be able to tell if a parse tree is acceptable just by looking at it, but users with a basic understanding of grammar can tell if “wedge” is a noun, which might be sufficient to rule out several parse trees. Clever use of indirect measurements can open even some of the most technical parse elements to user scrutiny, but care must be taken to only support questions that a user will be able to answer.

The last piece of the puzzle is to arrange the parse elements in a way that reflects the logic of the parse. GDE forms diagnoses by finding all combinations of completeness and incompleteness assumptions that are consistent with each other and the measurements taken so far. A diagnosis should indicate a fault (i.e., reject a completeness assumption) in just those cases where the system has not produced an acceptable interpretation of the sentence. When the system has produced such an interpretation, there should be no faults.

The most straightforward way to ensure this is to have a *root element* of the parse that is acceptable if at least one acceptable interpretation of the sentence exists and unacceptable otherwise. The acceptability of the root element should take all dependencies into account and cover all possible interpretations (i.e., combinations of choices).

One naive way to construct such an element would be to have one parse element for each complete interpretation of the sentence (i.e., each consistent combination of CNLU choices) and connect them all to the root. The justification structure would ensure that the root would be acceptable if any individual interpretation was acceptable and unacceptable if all of the interpretations were unacceptable.

This would work logically—the acceptability of the root element would capture the full space of interpretations—but it would come at the cost of a combinatorially prohibitive number of nodes. Given that there can be over a dozen choices in a choice set, that typical sentences have half a dozen choice sets, and that the label of each interpretation would have to take all relevant acceptability judgments into account, modeling the full set of interpretations individually, while conceptually simple, is too expensive.

Another naive approach would be to have a parse element for each choice set and connect them all jointly to the root element. Each choice set would be acceptable when at least one of its choices was acceptable, and the root would be acceptable when all of the choice sets were. This corresponds to the common intuition when disambiguating CNLU choices: that one must pick a choice from each choice set to construct a complete interpretation. Loosely, if there is a valid interpretation of each individual word, there is a valid interpretation of the sentence as a whole.

Unfortunately, this is not always the case. Some choices depend on other choices; they are only relevant when those other choices have been selected (in disambiguation) or are acceptable (in INLD). For example, the noun definitions of “wedge” are irrelevant in a sentence where “wedge” is used as a verb. The choices might exist, and INLD may ask the user about them, but once it is determined that “wedge” is a verb, they should have no effect on whether or not the sentence has a valid interpretation.

Typically, this type of dependency occurs with parse trees. The choice of a parse tree will enable some semantic choices and disable others, according to the parts of speech of the words in that parse tree and how the grammar rules pass the semantics up the tree. But dependencies between choices can also occur with adjectives, where the semantics of the adjective may depend on the semantics of the noun it modifies.

For example, consider the phrase “the large ducks”, which can be interpreted as an adjective modifying a noun (“the big waterfowl”) or a noun followed by a verb (“the large one dodges”). If “duck” is taken as a noun, the semantics of “large” will be either `(isa duck4150 (LargeFn Duck))` or `(isa duck4150 (LargeFn Duck-Meat))`. But these choices depend on the corresponding choice for “duck”; it is not coherent to say that “duck” refers to `Duck` when talking about the word “duck” but `Duck-Meat` when applying the adjective “large”. If the user were to say those two (and only those two) choices were acceptable, the sentence would not have an acceptable interpretation, even though one choice from each choice set is acceptable.

In short, it is not sufficient to pick one choice from each choice set independently to produce a complete interpretation. Dependencies between choices must also be taken into account, so that only the choices that are enabled are used to determine the acceptability of the root element.

Instead of these naive approaches, I use *factored interpretations* to correctly capture the space of possible interpretations by taking advantage of the dependencies between choices. The notion of a factored interpretation rests on the fact that the `enablesChoice` relation, which CNLU uses to capture dependencies between choices, is purely local. Choices can depend on any number of other choices (e.g., the semantics appearing in more than one parse tree), but only

one choice is necessary to enable them. There are no joint dependencies, and there are no cycles in the dependency graph. Thus, a choice is considered enabled unless all of the choices that would enable it have been ruled out.¹⁸

This allows the space of possible interpretations to be factored according to the choice dependency graph. The intuition is this: To produce a valid interpretation, select a choice from those with no dependencies (i.e., select a parse tree). Then, for every choice set where some of the choices are enabled by the selected choice, select one of those choices and repeat this process. Eventually, there will be no more choices left to make, in which case a valid interpretation has been found, or none of the available choices will be applicable, in which case the desired interpretation is missing. Because the dependency graph is acyclic, this process is guaranteed to terminate.

Factored interpretations reflect this process. Let the *factored interpretation at choice C* be a parse element which is acceptable if (1) the choice itself is acceptable and (2) an acceptable interpretation can be found using the above process, starting from the point where C is chosen. Essentially, the factored interpretation at C represents the choice selection subtree rooted at C. Because all choice dependencies are local, the acceptability of a factored interpretation can be considered independently of the choices used to enable it. All that matters is that (1) C can be chosen and (2) choosing C does not force the user into a situation where no acceptable choices remain.

To encode the latter situation, I introduce the concept of a *choice subset of S rooted at C*, where C is the choice under consideration and S is a choice set. The choice subset is the set of choices in S that are enabled by C. For example, in “the large ducks”, if C is the meaning of

¹⁸ Choices with no dependencies, namely parse trees, are always considered enabled.

“duck” corresponding to `Duck` and `S` is the choice set for “large”, the choice subset will only contain the choice for `(LargeFn Duck)`. The choice for `(LargeFn Duck-Meat)` will be ignored because it is not enabled by `C`.

For the factored interpretation at `C` to be acceptable, all of the choice subsets it enables must be acceptable; the process of picking new choices must be able to continue. To capture the recursive nature of this process, each choice subset is acceptable if at least one of the factored interpretations rooted at its choices is acceptable.

In this way, factored interpretations break up the dependency logic of choices and capture the space of possible interpretations. The root element, then, is just a factored interpretation for the sentence itself, representing the step in the process where no choice has been selected. It is acceptable if the root-level choice subset (i.e., all parse tree choices, since they have no dependencies) is acceptable. This is the case when the factored interpretation rooted at one of the parse trees is acceptable, meaning that acceptable semantic choices can be found for that parse tree. The construction process continues until it reaches choices that enable no other choices; their factored interpretation depends only on the acceptability of the choice itself.

An illustrative example can be seen in Figure 13, which shows the factored interpretation for one parse tree of “the large ducks”. Ovals represent facts in the ATMS, while triangles show their justifications. Green ovals represent the proposition that a parse element is acceptable (e.g., for interpretations) or that a set of possibilities is complete (e.g., for choice subsets). Red ovals represent unacceptability or incompleteness. Ovals with an outer circle are installed in the ATMS as assumptions, both so that they can be measured or faulted and so that the CATMS algorithm can use them to curtail unnecessary label propagation.

Note that this is only a fraction of the ATMS structure installed for this sentence. The actual choices for “large” are omitted from the left side of the diagram for reasons of space, as are the root interpretation for the sentence, the factored interpretation for the second parse tree, the part-of-speech measurements that determine whether Parse Tree #1 is acceptable, and the justifications that enforce mutual exclusivity between acceptability/unacceptability judgments and completeness/incompleteness assumptions. But the main patterns seen in the figure—the justification structures, the alternating layers of factored interpretations and choice subsets, and the hierarchical structure building to a single root—are the same ones used throughout INLD.

4.3 Debugging Strategies

In implementing the INLD debugging strategies for CNLU, I followed three principles: (1) questions should not require technical expertise from the user, (2) new structure should not make diagnosis intractable, and (3) semantics are more important than syntax.

The first principle rules out certain types of measurements that might be diagnostically useful but require too much expert knowledge. For example, it might be useful to know if a constituent should have a certain phrase type or exhibit a certain feature, but these measurements require that the user understand the difference between phrase types (e.g., a common noun phrase vs. a noun phrase) or the meanings of certain features. While there may be ways to request this information indirectly, such as by giving the user examples rather than using technical terms, I avoid adding any measurement points that require technical expertise.

The second principle rules out lines of attack that might eventually yield a diagnosis but which are computationally intractable or demand too much of the user. There is a vast amount of information that can be checked in a CNLU parse; considering too much of it at once can grind

the ATMS to a halt, while asking too many questions can wear out the user’s patience. The latter is especially important when considering that, for a correct parse, INLD is exhaustive. As there is no way to know that the last measurement won’t reveal a fault, INLD must continue until it has found a diagnosis or until it runs out of measurements to take. Thus, it is prudent not to add in measurements that are only circumstantially useful. The benefit of finding a niche problem with a parse must be weighed against the cost of asking those extra questions in every sentence where the problem does not occur.

The third principle suggests a natural boundary for the first two: Debugging strategies should prioritize semantic errors and deprioritize syntactic ones. In many cases, syntax is just a means to an end. It is rare that a CNLU application cares about a sentence’s parse tree, only the semantics it provides. Thus, while grammar rules and grammatical features are important, they should not be checked for their own sake. Rather, INLD should primarily focus on cases where the semantics of a sentence are incomplete, only delving into grammar to explain why a desired interpretation did not appear.¹⁹ Adopting this principle neatly circumscribes the large space of constituents in the chart and the nigh-infinite space of hypothetical parses that might otherwise be checked by an overzealous INLD system.

Broadly speaking, there are three possible outcomes for a parse element of interest: it reaches a top-level parse tree and joins the set of interpretations for the sentence (*success*), it is explicitly ruled out along the way (*failure*), or it is left on the chart, with no parent to make it part of a complete parse tree (*abandoned*). The initial model formulation deals with *successful* parse elements, only those that CNLU was able to use as part of a complete parse tree. But when these

¹⁹ Parts of speech are an exception. They are tractable, mostly non-technical, and diagnostically useful for ruling out parse trees, so my implementation includes questions like “What part of speech is ‘wedge’?” in its initial model formulation. But grammatical features and larger constituents are ignored.

sets of parse elements are found to be incomplete, INLD must examine the ones that failed or were abandoned to determine where the error lies.

Semtranses are a natural example. If none of the successful senses of the word “wedge” are acceptable, there are two possibilities: an acceptable sense failed or was abandoned, or there was no acceptable sense to begin with. The latter indicates a missing semtrans. But to prove it, we must ensure that none of the existing semtranses would have been acceptable if they had been successful. If there *is* a “sandwich” semtrans for “wedge” in the KB, the problem is that it does not reach a top-level parse tree to be presented as a choice, and the blame lies with whatever caused the correct meaning to be ruled out or abandoned.

My implementation handles this by tracing how CNLU parses sentences, mapping out the way semtranses are passed upward and transformed, and saving off a data flow graph that can be read from when necessary. All outcomes for the “sandwich” sense of “wedge” will be logged: the times it was ruled out, the constituents where it was abandoned, and the complete parse trees, if any, that contain it. My implementation also tracks valence patterns, which undergo similar processing as they are passed up the tree, and definitions, which are frequently abandoned if they cannot be used in a grammatical parse tree.

The general pattern for decomposition strategies, then, is to identify when a completeness assumption is violated, search for parse elements that *could* have been part of that set had they not failed or been abandoned, and install measurement points to ask the user if any of them are acceptable. If all of the failed or abandoned elements are unacceptable, or if there are no failed or abandoned elements to check, we can conclude that a fact is missing from the KB (a semtrans, a definition, etc.). If one is acceptable, the error lies with the reason it did not succeed: a faulty type check, a rule that should have fired, a missing valence pattern for a verb semtrans, etc.

This is where the approach runs into difficulty. Parse elements do not fail in one place; they fail in many. Every abandoned constituent containing a desired semtrans is a potential clue to the underlying error; maybe that constituent belongs in a complete parse tree. An acceptable valence pattern might fail in several different places due to various erroneous bindings. Without granular, expert knowledge of what the correct parse should look like, we do not have a singular error but a space of them.

Given enough time, no computational constraints, and an expert user, an INLD system could work its way through all of these possibilities, constructing the correct parse one piece of information at a time. But this approach violates all three guiding principles: It demands more expert knowledge than a user is likely to have, it is computationally and practically intractable, and it tends to get bogged down in syntactic minutiae.

In the spirit of providing a partial diagnosis when a complete one cannot be found, my INLD implementation does not attempt to find a singular reason why a desired parse element failed but instead saves the possibilities for an expert to examine later. It logs the constituents the parse element was abandoned on, the constituents where it was explicitly ruled out (e.g., due to a missing valence pattern or a type check), and the constituents where it picked up a piece of unacceptable semantics (e.g., due to a bad role relation expansion). My INLD system prints a summary of this information for the user while recording more detailed information to help an expert decide where to look first.

For example, Figure 14 shows the INLD report for “The man became disabled”. The completeness assumption for the choice set for “became” is faulted, and decomposition was able to identify that “became” *does* have an acceptable semtrans that was lost somewhere along the way, in at least one case due to not having a valence pattern it needed to continue.

```

For the sentence:
"The man became disabled"

These assumptions are faulted:
-Choice Set #36620 ("became") is complete.
-The semtrans outcomes for "became" are complete.

The verb semtrans for "Become-TheWord" with frame FN_Becoming produces no acceptable choices.
Here are some possible reasons why:
-It may be missing a valence pattern.
-It appears on 3 abandoned constits.

INLD found the following information:
Parts of Speech:
-"The" can be a determiner.
-"man" can be a noun.
-"became" can be a verb.
Semantic Types:
-"became" can mean a transformation.
-Words with no checked types: "man", "The"

```

Figure 14. INLD report for “The man became disabled”.

A consequence of being unable to check every outcome individually is that certain errors cannot be fully diagnosed. Among the errors of restriction described in Chapter 3, the CNLU implementation of INLD can currently diagnose missing lexicon entries (A1) and missing semtranses (C3), as they correspond to cases when no acceptable fact was found in the KB. Missing valence patterns (C2) are saved as potential errors when appropriate, but they cannot be confirmed without drilling down further. Likewise, errors related to type checking (C1, D1, or D4) are noted as a possibility when they rule out a desirable parse element, but they are not scrutinized further. Missing grammar rules (B1) are not currently addressed at all; they require extensive information about what constituents belong in the parse.

Implementing clever measurement strategies, giving the system more expert knowledge about what belongs in a correct parse, or sifting through possible outcomes with an LLM are all ways that INLD’s error coverage could be expanded past the current barrier.

4.4 Evaluation

To evaluate my INLD implementation, I tested it on 50 sentences sampled from the Choice of Plausible Alternatives corpus (COPA; Roemmele, Bejan, & Gordon, 2011). COPA is a

dataset that presents short textual scenarios to an AI system and asks it to judge which of two outcomes is more plausible. The dataset is intended as a test of open-domain commonsense causal reasoning, but it is an attractive starting point for debugging CNLU. The sentences tend to be short, so most of them can be run through INLD without causing ATMS blowup, while the open domain makes them a nice sampling of text CNLU should be able to handle.²⁰

The full COPA corpus consists of 1500 sentences. To avoid wasting time on sentences that CNLU or INLD decidedly cannot handle, I prefilter the corpus by removing any sentence that takes longer than 30 seconds for CNLU to parse or longer than 5 seconds for INLD to set up the ATMS. Performance of both systems depends on the hardware on which they run, but these cutoffs ensure that most of the remaining sentences can be run in a tractable amount of time. Among the sentences tested this way, roughly 75% passed the filter, with 4% failing because the parse timed out and 22% because INLD setup timed out. Raising the INLD setup timeout to 30 seconds reduces the failure rate to 18%, but a sentence that is slow to set up may be prohibitively slow to diagnose. Thus, enforcing a strict timeout for setup is advantageous.

To assess my CNLU INLD implementation, four annotators diagnosed the 50-sentence COPA sample. The annotators were all familiar with CNLU, but they did not have access to its internal state while debugging, only the information that was exposed through the NL interface. Three annotators were able to complete the entire set of sentences, while the fourth, working on a weaker laptop, had five sentences time out during setup with a 60-second threshold.

²⁰ It's worth noting that the difficulty of INLD varies according to the complexity of the dataset and the NL system it is based on. Ideally, running INLD over a corpus yields corrections that "solve" the corpus for the parser. Unlike ML benchmarks, the appropriate measure is not how difficult the dataset is in terms of linguistic complexity but whether it can be used to make meaningful debugging progress. Thus, while the COPA corpus is linguistically simple, it allows us to systematically test and debug CNLU in ways that have not been tried before.

Out of the 50 sentences, 10 were unanimously found to be faulted and 15 were found to be faulted by at least three annotators (of which 3 sentences timed out for the fourth annotator, for a total of 13 unanimously faulted sentences among annotators who completed them). From the perspective of unfaulted sentences, 14 sentences were unanimously found to contain no faults, while 16 were found to be faulted by only one annotator.

Notably, missing definition errors (triggered by an unknown word or a known word not having an acceptable part of speech) were rare. There were only 3 instances of them total, none of which were attested by more than one annotator. This indicates that CNLU’s lexical coverage is adequate for the COPA corpus, even if there are holes to be patched in its semantics.

I assess the performance of INLD two ways. First, in Section 4.4.1, I measure inter-annotator agreement (Artstein, 2017) to determine how consistent INLD is across users. Imperfect verbalizations, differing linguistic intuitions, and other types of variability can have a large impact on the output of the system. While noisy diagnoses are merely an inconvenience if the purpose of the system is to help experts find and diagnose bugs, consistency is important when dealing with the full INLD pipeline, untrained users, or automation.

Second, in Section 4.4.2, I inspect the diagnosed errors manually and assess whether they are reasonable. As there are multiple ways to interpret an ontology, my judgments should not be taken as definitive; another CNLU developer might justifiably disagree. But a sentence will often have a clear-cut error that INLD may or may not have been able to diagnose. Thus, I label the faults found by the four annotators as correct or incorrect, use them to calculate success rates, and comment on the viability of INLD-aided diagnosis.

I follow these analysis with a pilot experiment that explores automating INLD with an LLM oracle (Section 4.4.3), then conclude the chapter with a brief discussion of my findings (Section 4.4.4).

4.4.1 *Agreement*

Measuring inter-annotator agreement for INLD poses a few challenges. Sentences may have more than one fault, and their faults may differ in granularity depending on how far INLD was able to drill down. Two diagnoses may agree that, say, a choice set is missing a meaning but differ in whether a ruled-out semtrans is acceptable. Lastly, because INLD selects measurements adaptively, not every annotator will be asked the same set of questions. These challenges are not insurmountable, but they must be worked around with care.

I measure three forms of agreement: (1) for each sentence, agreement on whether it was faulted or not; (2) for each choice set in each sentence, agreement on whether it was complete; and (3) agreement on the acceptability judgments of questions posed to at least two annotators. The first indicates how reliable INLD is at flagging buggy sentences. The second indicates how reliable it is at identifying symptoms. The third indicates how reliable user judgments are. In all three cases, I report the modified version of Fleiss’s κ (Fleiss, 1971) from Gwet (2014) which allows variable numbers of annotators for each question. As agreement on specific error diagnoses is both hard to measure fairly and downstream of the other types of agreement, I reserve discussion of error diagnoses for my manual analysis below.

The results were as follows. For sentence faults, $\kappa = 0.48$. For choice set completeness, $\kappa = 0.47$. For measurements, $\kappa = 0.24$. These results indicate a moderate positive correlation but not a perfectly reliable one. INLD exhibits reasonable agreement at identifying buggy sentences and the incomplete choice sets responsible for them, but agreement for measurements is low.

One might expect²¹ measurement agreement to be the limiting factor on the other types, as faults are downstream of the questions used to diagnose them, but this does not seem to be the case. The resolution to this paradox is that most measurements are not critical for diagnosis. If a choice set is complete, it only matters that *one* of its choices is acceptable, not *which* one. Users may disagree as to which senses are acceptable for a word, but as long as they both find one that is acceptable, the discrepancy will not matter to the final diagnosis.

To analyze the variability of these results, I calculated agreement for all pairs and trios of annotators. The highest overall agreement for three annotators was $\kappa = 0.48$ for sentence faults, $\kappa = 0.45$ for choice set completeness, and $\kappa = 0.65$ for measurements, although one trio had the interesting spread of $\kappa = 0.63$, $\kappa = 0.60$, and $\kappa = 0.00$, respectively. The lowest overall agreement for a trio of annotators was $\kappa = 0.38$ for sentence faults, $\kappa = 0.42$ for choice set completeness, and $\kappa = 0.05$ for measurements.

The highest agreement for any pair of annotators was $\kappa = 0.66$ for sentence faults, $\kappa = 0.59$ for choice set completeness, and $\kappa = 0.73$ for measurements. The lowest was $\kappa = 0.24$ for sentence faults, $\kappa = 0.33$ for choice set completeness, and $\kappa = -0.43$ for measurements.

These numbers should **not** be taken as a substitute for the overall agreement, but they do shed light on what range of outcomes is possible, assuming a similar annotator pool to the one here. Agreement on sentences and choice sets stays fairly constant across any three annotators, but agreement on measurements varies wildly. This dichotomy is even more stark when considering pairs of annotators. Even a pair with $\kappa = -0.43$ for measurements showed slight agreement on the other two measures, while one noteworthy pair had a spread of 0.52/0.50/-0.52. The sharp decoupling between measurement agreement and the other two indicates that it is not a

²¹ I expected.

useful measure of downstream performance. While a high measurement agreement is desirable, a low one does not necessarily impact agreement on the outputs that matter.

Meanwhile, the other measures stay fairly consistent in a range of $\kappa = 0.38$ to $\kappa = 0.63$ for any trio of annotators. Even pairs of annotators, which show higher variability, do not dip below a weak positive correlation ($\kappa = 0.24$ for sentences and $\kappa = 0.33$ for choice sets), with most pairs also falling in the $\kappa = 0.40$ to $\kappa = 0.60$ range for both sentences and choice sets.

The high ceiling on agreement between pairs of annotators also shows what INLD can do under the right conditions. The best pair achieves a κ spread of 0.66/0.59/0.73, and their *raw agreement* scores were 0.86 for sentence faults, 0.94 for choice set completeness, and 0.89 for measurements. While Fleiss's κ is a useful measure of agreement that accounts for the baseline difficulty of the task, raw agreement matters more when looking for a consensus diagnosis. An 89% agreement rate on acceptability judgments and a 94% rate on choice set completeness are a strong foundation to build on, while 73% agreement on individual measurements shows that there is a high ceiling there as well.

One should not put too much stock in these specific numbers, but they provide evidence for three useful findings. (1) The task is well-formed, with reasonable rates of inter-annotator agreement. (2) The task is relatively insensitive to outliers. Agreement varies depending on the annotators selected but rarely drops significantly. (3) The task is robust to disagreements about questions. Even a strong *negative* correlation in answers does not always mean that the faults found by two users will differ more than normal. Changes to boost agreement (e.g., clearer verbalizations) are welcome but are not necessarily a bottleneck for debugging performance.

While inter-annotator agreement is a worthwhile measure of consistency, it does not say whether the diagnoses themselves are accurate. I turn to this question next.

4.4.2 Analysis

To determine the quality of the diagnoses produced by INLD, I manually examined all 95 faults found by any annotator across the 36 sentences that had them. I assessed these faults from the perspective of a CNLU developer: Are they a mistake by the system or a misconception by the user? When in doubt, I sided with the user on the grounds that a debatable error is still worth investigating.

The specific numbers should be taken with a grain of salt, as they depend on the number of faults INLD takes to diagnose a particular type of error, which can be variable. The current implementation typically produces two faults per error, an incomplete choice set (symptom) and a fault explaining it (error), but there are corner cases that only yield one fault, and in principle there can be more. For simplicity, I analyze faults as-is and do not try to bin or normalize them.

My primary results can be seen in Figure 15, a chart of fault correctness broken down by the number of annotators that found them. Faults reported by at least one annotator were correct

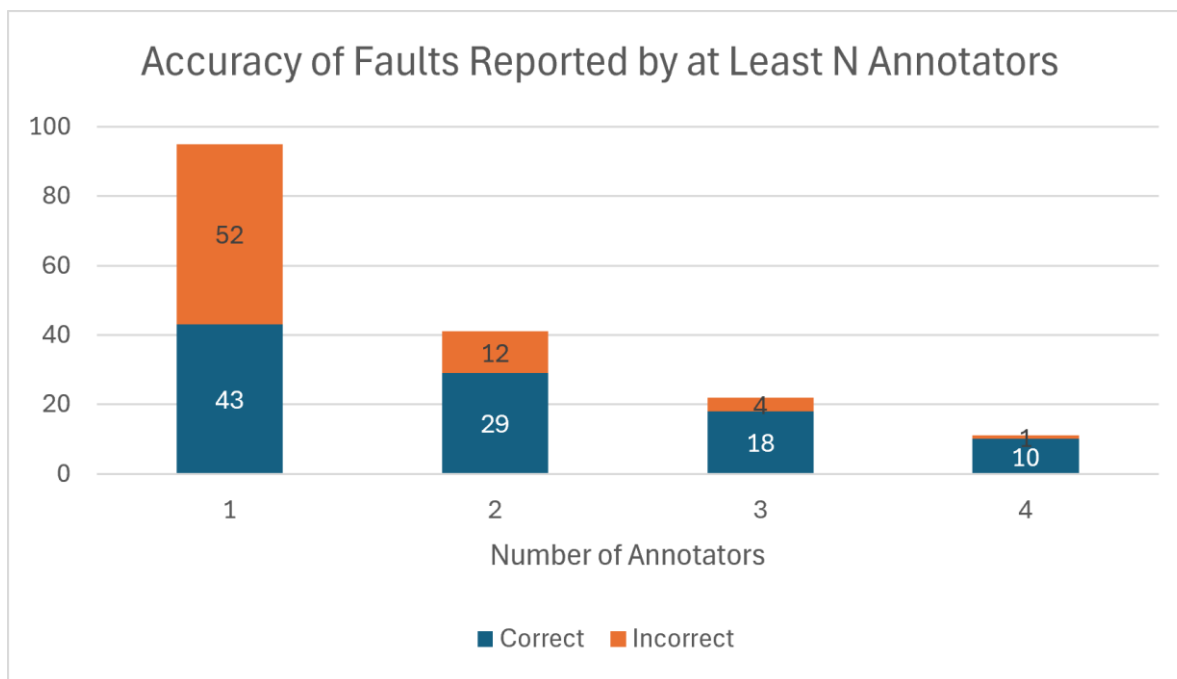


Figure 15. Fault correctness by minimum number of annotators who found them.

45% of the time (43 correct out of 95 faults found). Faults that were corroborated by at least two annotators were correct 71% of the time (29 out of 41), while increasing the cutoff to three annotators produced an accuracy of 82% (18 out of 22). Faults that all four annotators agreed on were correct 91% of the time (10 out of 11).

The chart shows a clear tradeoff between the number of faults found and the error rate. Accepting the faults found by any annotator casts the widest net but yields a high error rate, while corroborating faults with multiple users reduces the error rate sharply at the cost of finding fewer faults overall. This pattern is expected, but it is useful to confirm that INLD behaves as intended. Given that bugs are easier to check than they are to find, even a 55% false positive rate represents a favorable tradeoff for developers, and corroboration by multiple users can reduce that as needed. The more users who are able to search for bugs, the more will be found, and having a preliminary diagnosis makes it easier for developers to confirm and correct them.

Qualitatively, I observe that most of the corroborated faults were sensible. CNLU shows good semantic coverage for the COPA corpus, but annotators found 7 words with missing semtranses and a further 10 where an acceptable semtrans could not be applied correctly, usually due to a missing valence pattern. This indicates that INLD was able to render the semantics of the parse into clear enough English that symptoms could be noticed and errors diagnosed. As noted before, there were only 3 faults for missing definitions, none of which were corroborated by other annotators and none of which were found correct upon review. Among uncorroborated faults, spurious faults were common. These could be the product of differing judgments, unclear verbalizations, or input errors. Again, corroboration helps alleviate all of these issues.

As an illustration of the kinds of errors found, here are a few examples. The sentence “The police dropped the case” uses a sense of “dropped” that CNLU doesn’t currently have. The

sentence “The driver talked the whole way” appears to be missing a valence pattern to relate “the whole way” to “talked”, which points to a broader linguistic construct that should be handled systematically (“She complained the entire trip”, “He walked the last hour”, etc.). There were a few cases where a bad verbalization caused confusion, as with the role relation `emitter` in the context of `Snoring`, whose verbalization implies the emitted thing is an object. These types of errors may be mundane, but they are the bread and butter of CNLU development, and each one is an incremental step towards broader coverage.

4.4.3 *Automation*

Advances in LLM technology introduce a new possibility for INLD: using an LLM as the natural language oracle instead of a human. Assuming the LLM can reliably answer questions about language, an LLM oracle could make INLD fully automatic. Given a sufficiently robust INLD system, an LLM could continue diagnosing (and one day correcting) errors indefinitely. This would be a tremendous boon for developers of hand-curated NL systems. Once set up, the system could improve itself continuously as long as compute was available.²²

As an initial assessment of this possibility, I replicated the above experiment with the 32B parameter version of DeepSeek-R1 (Guo, et al., 2025) running on Ollama.²³ I queried the model with the same questions as in the human case, but each question was prepended with “Consider the following sentence: <text>” to provide context. Questions were asked separately, rather than in a chat session, and output was constrained using JSON schemas to ensure the

²² In the extreme case, this takes the “hand” out of “hand-curated”, but such systems would still have the advantage of explicit, inspectable, and correctable linguistic knowledge that evolves incrementally, rather than through retraining. Thus, even with an LLM largely taking over the role of expert, this setup differs from both pure neural approaches and statistically induced grammars. But this level of automation is a ways off.

²³ <https://ollama.com/library/deepseek-r1:32b>

model produced valid answers. Questions were asked zero-shot, with no finetuning for the task and no examples given in the prompt itself.

Over the 50 sentences, DeepSeek-R1 found 49 faults in 25 sentences. 17 of these faults were correct, for a correctness rate of 34%. This is well below the rate of any single annotator (51%, 53%, 63%, and 72%, respectively) and the group as a whole (45% for faults found by any annotator, and 91% for those agreed upon by all annotators). The LLM’s agreement with the annotator with the highest correctness rate was $\kappa = 0.39$ for sentence faults, $\kappa = 0.36$ for choice set completeness, and $\kappa = 0.56$ for measurements. These numbers fall within the low end of the range of agreement for other annotator pairs.

The performance of the model is likely too unreliable for a fully automatic setup, but it shows promise. The ability to mine for CNLU errors automatically would be very useful for expanding coverage, and even a high false positive rate might be acceptable if the diagnosed errors are easy for the developers to check. Moreover, the LLM could be used to find sentences with suspected errors for users to check. In this way, users only have to examine sentences that are likely to contain an error, and the developers only have to investigate errors corroborated by a human. This is just one of many ways an INLD setup could put even an imperfect LLM to use.

Qualitatively, DeepSeek-R1 sporadically makes mistakes that a human would not. For instance, it answered that the word “coffee” did not mean “coffee”, that “vehicle” did not mean “self-powered vehicle”, and that “students” did not mean “student”. While the verbalizations of these concepts are perhaps tautological and could use further clarification, they are clearly the right choices in the context of their respective sentences. As with other LLM judgments, it is hard to say whether these mistakes were a product of the awkward wording, a lapse in adhering to the list-of-numbers answer format, or simply random chance. DeepSeek-R1 also sometimes

had trouble with role relations that connected an event (verb) to one of its arguments. This could indicate that role relation verbalizations should be clearer or that the model should be coached in Neo-Davidsonian representations as part of its prompt.

While DeepSeek-R1 represents a convenient baseline for what a recent, moderately sized model can do, this is by no means the optimal setup for automating INLD. Prompt engineering, larger models, few-shot learning (Brown, et al., 2020), chain-of-thought reasoning (Wei, et al., 2022), multiple runs, multi-model ensembles, and many other optimizations may yield improved performance and provide a more reliable engine for INLD. As INLD matures and becomes ready for automation, it becomes worthwhile to optimize its LLM oracle more aggressively.

It is also worth noting that using an LLM opens the door to certain options that are not possible with a human. While the model may not be an expert in the particular NL system at hand, it may have a general knowledge of computational linguistics that surpasses that of the average non-expert user. Combined with term definitions in the prompt, this may allow the LLM to answer more technical questions that give INLD more leverage for debugging.

Likewise, the only limits on an LLM’s question-answering capabilities are time and the cost of computation. Thus, the exhaustive decomposition discussed in Section 4.3 may become viable. Whereas it would be poor use of a user’s time to laboriously hunt down each outcome of a semtrans in search of a bug, an LLM’s time cannot be wasted.²⁴ This even suggests a hybrid setup where, given sufficiently cheap LLM access, the LLM does a deep dive on the sentence, finds an error, and then passes the results back to a modified INLD setup that asks the user to confirm the LLM’s key judgments instead of starting the sentence from scratch. Smart use of an LLM can open up new horizons for INLD.

²⁴ Only a researcher’s money.

4.4.4 Discussion

This experiment shows that INLD succeeds in its basic goal of allowing a user to help debug an NL system without inspecting its internal representations. The current implementation is limited to missing word definitions, missing semtranses, and partial information about missing valence patterns and bad type checks. It also only covers the first two stages of the INLD pipeline, as correction poses a separate set of challenges, especially with a text-only interface. The experiment shows moderate agreement between annotators, passable fault correctness rates for individual annotators, and impressive correctness rates for faults corroborated by multiple annotators. Initial testing with an LLM shows promise. It performs worse than humans but still well enough to provide a useful signal. Further testing is needed to optimize its usage.

While this experiment is limited to the relatively simple sentences of the COPA corpus, it demonstrates an important use case for INLD: improving coverage. Even in COPA, there are words that are missing semantics, verbs that are missing valence patterns, and other anomalies. Patching these holes in CNLU’s linguistic knowledge is a major way to make CNLU more reliable and prepare it for future applications. As INLD’s capabilities are expanded to better handle grammar errors, it will be worthwhile to run it on more complicated sentences to test CNLU’s handling of linguistic phenomena like modals, conjunctions, and negation.

Before moving on from INLD, it is worth discussing some of the limitations uncovered during this experiment. One of these is the limits of verbalization. While most questions were verbalized well enough for diagnoses, there were a few cases where the verbalization was incoherent or ambiguous.

For example, consider the question in Figure 16. This is an unfortunate case where verbalization produces an ambiguity that clarification fails to correct. The underlying concepts

are `WorryingAboutSomething`, `(CausingFn (ExperiencingEmotionFn Concern))`, and `(ExperiencingEmotionFn Concern)`, respectively. Ideally, these would be verbalized as “worrying about something”, “causing someone to experience concern”, and “experiencing concern”, which would capture the subtle distinction between feeling concern in general and worrying about something in particular.

What is "worried" in this sentence?

- 1) worrying**
- 2) causing worrying**
- 3) worrying (an action)**

Figure 16. Question about “worried” in “She worried she would catch the flu.”

Instead, both `WorryingAboutSomething` and `(ExperiencingEmotionFn Concern)` have the same `prettyString`. And because the only parent collection that can be used to distinguish one from the other is `Action`, the user is left without any meaningful guidance as to what the system is asking. Overlap between verbalization is to be expected in an ontology the size of `NextKB`, especially given the subtle distinctions between certain concepts and the comparative lack of testing for the generation knowledge used by `verbalize` compared to the parsing knowledge used for `CNLU`. This is an expected outcome of maintaining a broad-coverage KB, and these types of ambiguities can be cleaned up over time.

But it still causes problems for INLD. This specific question is not likely to do harm—picking any of the three options will make the choice set for “worried” complete—but there are other scenarios where a confusing or wrong verbalization could derail the entire diagnosis (e.g., by making an acceptable word sense appear unacceptable, leading the system to erroneously try to find the missing word sense instead of moving on to other measurements). Developing a more

robust clarification system would help clean up this source of noise, make INLD more robust as a whole, and help flag `prettyString` conflicts for future revision.²⁵

Another noteworthy limitation is ATMS bloat. Even prefiltering the COPA corpus for sentences that are fast to set up, five sentences that ran quickly on one laptop timed out when run on a weaker one, showing that performance does not decay gracefully. Likewise, the high rate of INLD timeouts during prefiltering (~22%) suggests that improving the efficiency of INLD is an important area of future work.

Ideally, the algorithm should be smart enough to manage its own resource usage, without any preprocessing, and decomposition is a natural fit for this. Rather than considering the entire sentence at once, INLD could construct the ATMS structure for one portion of the sentence at a time, only moving onto the next part of the sentence if no faults are found. Decomposing too aggressively will interfere with the GDE algorithm’s ability to pick the best measurement to take, as not all of the measurements will be available right away, but the improvement in performance should make it a worthwhile tradeoff. Nakos, Kuthalam, and Forbus (2022) discuss simplifying sentences as a diagnosis strategy, but a similar approach might work to maintain tractability. In any case, care must be taken to keep the ATMS structure lightweight, so that the problem does not bog down as the rest of the sentence is brought in.

Finally, I observe that there is a mismatch between the needs of ontologization and the needs of communication. NextKB contains many concepts that are named similarly, verbalize similarly, and can only be told apart by technical distinctions in their comment strings. This is a fine approach for reasoning, where subtle distinctions matter, but it is an imperfect fit for the way

²⁵ Specifically, the system should test whether generated options are ambiguous. Preliminary experiments with token overlap were unsuccessful, but using an LLM to screen for ambiguity would be an interesting solution. Ideally, the clarification system should also have more remedies than mentioning distinguishing parent collections.

people use language, where intuitions matter more than formal definitions and the boundaries between concepts are drawn in different ways. Clarification techniques can help in some cases, but having more language-friendly, user-facing knowledge—concept examples, distinguishing questions, paraphrases, etc.—would help make NextKB more accessible to non-expert users via systems like INLD.

5. Generative Example-Driven Error Mining

While INLD is useful for debugging suspected errors, it is inefficient at finding them in the first place. Testing sentences blindly means sifting through the correct sentences to get to the erroneous ones, a process slowed by the fact that correct sentences require answering INLD’s entire suite of questions. The more complicated the sentence, the slower the computation and the more questions have to be asked. While automation with LLMs might alleviate these problems to some degree, INLD is a framework designed for depth rather than breadth. Furthermore, it focuses on *errors of restriction*, which prohibit a correct interpretation from being found, as opposed to *errors of permission*, which allow ungrammatical or nonsensical parses.

This suggests a complementary framework, one that seeks out errors in bulk rather than drilling down on specific sentences. Such a framework exists. *Error mining* automatically identifies gaps in a parser’s coverage by running the parser over a text corpus and identifying *n*-grams or other features that are commonly associated with fragmented parses (de Kok & van Noord, 2017; van Noord, 2004). Similar work (Goodman & Bond, 2009) uses round-trip generation to identify gaps in an NL system’s generation capabilities.

The key insight of this work is that the system does not need to understand why each parse fragmented to extract useful information from them. Even if the system lacks the insight to attribute blame to a specific word, phrase, or rule that the parser handles incorrectly, it can gather statistics—typically in the form of a *suspicion score*—that suggest which linguistic forms the parser most frequently has trouble with. These suspected errors can then be given to an expert for further investigation, akin to the division of labor between experts and non-experts in INLD.

This is a very elegant approach, but it poses a challenge for NL systems that use semantics. Syntactic systems have a clear error signal that error mining can exploit: “Does the

sentence parse?” With the use of annotated treebanks (such as the influential Penn Treebank; Marcus, Santorini, & Marcinkiewicz, 1993), the error signal becomes even more precise: “Does the sentence parse *correctly*?”

But with semantics, the picture is not so clear. There is no automatic signal analogous to a fragmented parse. The fact that a semantic parser manages to produce an interpretation of a sentence does not mean much by itself. What matters is whether the interpretation is correct, and that is a difficult question to answer automatically. The problem is complicated by the fact that “correctness” depends on the representations used in the interpretation and what ontology they are grounded in. Semantic ontologies are larger and more complex than the fixed sets of phrase types employed by syntactic parsers, so the output of a semantic parser is a broader space to check. Ontologies also vary dramatically in how they carve up the world, making them difficult to compare directly to one another. All of this means that there is no simple, reliable, and fully automatic indication when a semantic parser interprets a sentence the wrong way.

Thus, another type of signal is necessary to mine for semantic errors. One approach would be to use semantically annotated corpora (e.g., OntoNotes; Hovy, Marcus, Palmer, Ramshaw, & Weischedel, 2006) as ground truth. Then the output of the semantic parser can be checked against the provided annotation, and any discrepancies can serve as a signal for error mining.

However, there are two problems with this approach. One is finding semantically annotated corpora to use. Human annotation is expensive, and the more detailed the semantics, the more technical expertise is required to produce it. Many of the existing datasets in this space focus on narrower tasks, such as word sense disambiguation (Bevilacqua, Pasini, Raganato, & Navigli, 2021) or relation extraction (Detroja, Bhensdadia, & Bhatt, 2023), which are useful for

certain Natural Language Processing applications but fall short of the complete semantic interpretations a semantic parser is meant to output.

The other problem is that semantic annotations are fixed: They do not evolve with changes in the ontology or representation language. This works for well-established, widely-adopted ontologies like WordNet (Miller, 1995) or ConceptNet (Speer, Chin, & Havasi, 2017). In this case, the ontology serves as a “good enough” standard to use, and any errors or gaps in coverage can be treated as out of scope. But for evolving ontologies such as NextKB and evolving NL systems such as CNLU, a fixed corpus of annotations may quickly become out of date. Errors in knowledge representation may not be obvious at first, and the need to correct them makes the “right” interpretation of a sentence a moving target.

To address these issues, I propose *Generative Example-Driven Error Mining* (GEDEM; Nakos, Demel, & Forbus, 2025), a framework for semi-automatically mining for likely errors in hand-curated NL systems. The basic idea is to generate multiple sentences for the same semantics and use *paraphrase and grammaticality judgments* as an error signal. Any sentences that are ungrammatical or are not meaning-preserving paraphrases of the others indicate an underlying error in the system’s linguistic knowledge. Unlike INLD, where the appearance of a symptom is used to kick off in-depth diagnosis, GEDEM gathers statistics over a corpus of examples to suggest likely errors. In this way, GEDEM can find errors in the linguistic resources of a semantic parser without needing to dive deeply into the semantics of individual sentences. While not fully automatic if a human is providing the judgments, the only inputs are paraphrase/grammaticality judgments, as well as any additional knowledge needed to sample the semantics (e.g., prototypical role fillers).

For example, the left side of Figure 17 shows the semantics for CNLU’s interpretation of “The man owns the cat”. The right side of the figure shows three other sentences that parse to the same semantics. Two of these are good paraphrases of the original sentence, but the third, “The man lacks the cat”, is not. This indicates an error in CNLU’s knowledge. In this case, it is an erroneous semtrans: the verb “lack” maps to the concept `OwningSomething`. When parsing “The man lacks the cat”, CNLU will produce the nonsensical `OwningSomething` as one of its choices. When generating a sentence from the semantics, CNLU will produce “The man lacks the cat” as though it properly conveyed the intended meaning. While more reasoning may be needed to localize the underlying error, a single user judgment—that one of the sentences is not like the others—is enough to suggest that an error is present.

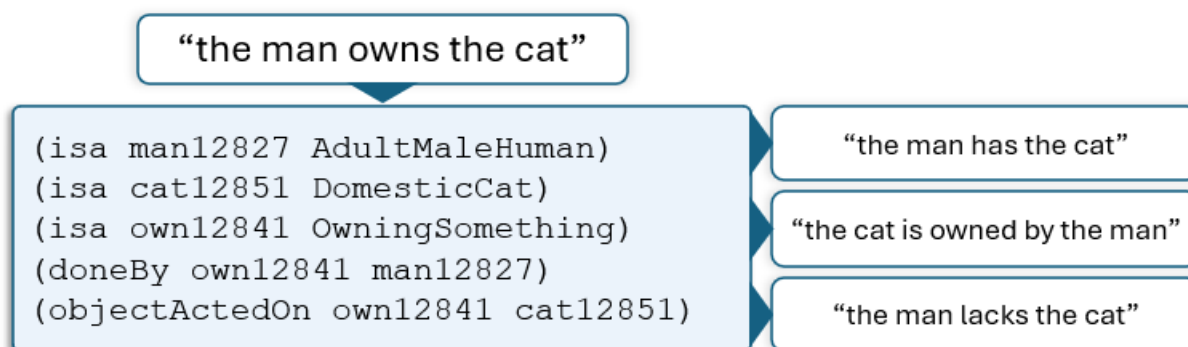


Figure 17. CNLU semantics and paraphrases for “The man owns the cat”.

This highlights one of the core ideas of GEDEM: If the same linguistic knowledge is used for both understanding and generation, generation can be used to find errors that are not apparent from understanding alone. An error that goes unnoticed during understanding may become obvious when it causes bad sentences to be generated (errors of permission), while gaps in the system’s generated output correspond to sentences that cannot be understood properly (errors of restriction). They are two sides of the same coin: minor issues in understanding show up as major problems in generation, and vice versa.

In the remainder of this chapter, I discuss the GEDEM framework and how it applies to CNLU (Section 5.1). I then introduce *say*, the NLG component of CNLU (Section 5.2) and discuss its use for GEDEM. Finally, I discuss the results of an experiment using GEDEM to identify likely errors in CNLU verb semtranses and valence patterns (Section 5.3).

5.1 The GEDEM Framework

GEDEM works by sampling instantiated semantics from the space of interpretations that can be produced by a semantic parser, generating a set of sentences for each sample, and having a user judge which of the sentences are ungrammatical and/or a bad paraphrase of the others. Performed over enough samples, this provides an error signal that can be used to identify likely errors in a system's linguistic knowledge. These can then be given to an expert to be confirmed and corrected, or they can be passed to a debugging tool like INLD for correction by a non-expert user, if a tool that covers the appropriate error type is available.

GEDEM is designed to work with a hand-curated NL system that uses the same linguistic knowledge for understanding and generation. That is its simplest and most powerful use case, where improperly generated sentences point to errors of permission in the system's linguistic knowledge, and it is the use case which I will be exploring here. But in principle, the technique also applies to hand-curated NLG systems without a parsing component. As long as the system can produce several pieces of text for the same semantics, they can be compared to one another to find likely errors.

The same goes for semantic parsers with no text generation component. All that is needed in that case is a reliable way to find sentences that map to the same semantics. For example, a set of templates could be used to cover common sentence types, then words can be

swapped out to find sentences that parse to the same interpretation. While limited by the generation capabilities of this *ad hoc* NLG system, in a pinch, it would suffice to reveal certain types of errors using paraphrase mappings.²⁶

5.1.1 *The Generative Story*

One of the core components of GEDEM is the *generative story*²⁷ used to sample semantics from the space of possible interpretations. This component is flexible; any random function that generates valid semantics will suffice. However, the better coverage the generative story achieves, the more errors it will be able to find. Selecting a more powerful generative story will produce more varied semantics, and it will better test the system's breadth of linguistic knowledge. Selecting a narrower generative story will help focus GEDEM on specific types of errors.

For CNLU, the generative story can be quite complex. On top of basic semantics for nouns and verbs, quantifiers and modals can yield various types of higher-order structure, modifiers like adjectives, adverbs, and relative clauses can tack on additional facts about any discourse variable, and conjunctions can make interpretations even more complex. This type of representational richness is to be expected in a broad-coverage semantic parser, but it raises the question of where to even begin.

I opt for a simple generative story designed to mine for errors where they most need to be found: semtranses and valence patterns. The process is as follows: sample a verb semtrans at

²⁶ Conceivably, GEDEM could even apply to parsers with no semantics at all. As long as sentences can be sampled from the grammar, grammaticality judgments can be correlated with the rules used to generate them. Without semantics, this approach will produce nonsensical sentences, but grammatical errors can still be identified. This reduces to the early NLG work of Yngve (1961), where sentences were sampled to reveal errors in a grammar.

²⁷ I use this term in the Bayesian sense: an account of how a random process yields values for an observed variable. In the case of GEDEM, the observed variable is a semantic interpretation suitable for generation and the random process is a method for sampling it from the system's linguistic knowledge.

random, pick one of its valence patterns,²⁸ and instantiate its role relations with *role fillers*, entities that typically play a particular role in an event of that type. The result is a piece of semantics describing a single Neo-Davidsonian event and its participants, corresponding to a simple declarative sentence such as “The man owns the cat.” or “Jill gave Jack the book.”

Role fillers are an additional type of knowledge the system needs solely for the purpose of generating plausible examples. While role relations in NextKB have type constraints that limit what arguments they *can* take (e.g., `performedBy` can only take an `Agent-Generic` as its second argument), NextKB does not have information about what arguments they typically *do* take. Furthermore, the same role relations are used for different types of events where their typical fillers might vary. For example, `instrument-Generic` appears as a role relation in valence patterns for over 200 verbs, but the instruments typically used for, say, “baptize” and “pulverize” are very different. Thus, to avoid generating absurd or poorly calibrated examples, the system needs to know what types of entities typically participate in which kinds of events.

I address this problem with a knowledge capture loop that asks either the user or an LLM for typical role fillers for a verb one at a time. The system picks a verb and one of its valence patterns, then loops through the role relations in the pattern and asks about each of them in turn. For example, the verb “laps” has a valence pattern that has the role relations `performedBy`, `consumedObject`, and `from-Generic`. These might be respectively filled by “the kitten”, “milk”, and “a saucer”, corresponding to a sentence like “the kitten laps milk from a saucer”.

The prompt for eliciting a role filler can be seen in Figure 18. As this prompt is intended for use with an LLM, it is verbose and contains explicit task descriptions and reminders. Testing

²⁸ For simplicity, and to accommodate the current limitations of say, I only select valence patterns that use `:SUBJECT`, `:OBJECT`, and `:OBLIQUE-OBJECT` and which take noun phrases and prepositional phrases. Expanding this process to other grammatical functions and phrase types is an area for future work.

has shown this scaffolding to be necessary to produce acceptable results, but further refinement of the prompt is possible. Role fillers are elicited one at a time, rather than in a chat session, but the previously found fillers are provided in the prompt for consistency. Elicited filters are checked for suitability, then saved to the KB for future use.

```
I am trying to come up with an example sentence for the word "laps", meaning drinking (a consuming).
I am filling in its arguments one at a time.

One of the arguments that "laps" can take is the generic origin.
I am looking for a short, simple, specific phrase that can be used as
the generic origin of "laps" in a sentence.
It should fit with the following other arguments:
    performer = "the kitten"
    thing consumed = "milk"
What is a short, simple, and specific phrase that can be the generic origin of "laps"?
Remember: Just give me a short noun phrase. Don't try to use it in a sentence.

(Enter a string, :help, or :exit.)
> a saucer|
```

Figure 18. Example of eliciting a role filler for the word “laps”.

This approach to generating role fillers differs from the one described by Nakos, Demel, and Forbus (2025), which used templates to elicit entire sentences of role fillers at a time. This approach has the advantage of prompt economy (i.e., multiple examples could be found with a single request) and example diversity (because the LLM is generating several sentences at once, rather than gravitating towards the same default topics with each fresh request), but it lacks fine-grained control. The current approach elicits role fillers incrementally and checks them as it goes, ensuring most issues are caught early rather than propagating through the rest of the pipeline. Specifically, fillers are discarded if they are longer than five tokens, if CNLU fails to parse them, or if say fails to regenerate them. The length cutoff reduces parsing complications and curtails unnecessarily specific examples. The other two checks ensure that the filler is suitable for GEDEM, namely that (1) it can be parsed to produce semantics and (2) those semantics can be used to help generate example sentences for the user to check.²⁹

²⁹ Both during checking and when role fillers are used to sample interpretations, role fillers are parsed and disambiguated using a deterministic strategy based on lexicographic ordering. For the purposes of GEDEM, it does not matter that CNLU finds the *correct* meaning of the phrase, only that it finds one it can use in an interpretation.

The final step is to construct a semantic interpretation that can be used for generation. Each set of role fillers provides the basis for one interpretation. The role fillers are conjoined with the verb using a simple template to form a sentence for CNLU to parse. GEDEM selects appropriate choices to ensure the sentence’s interpretation is consistent with the desired valence pattern, picks the remaining choices deterministically, and extracts the final interpretation as a semantic expression suitable for use with say. Every sentence generated from this interpretation will be a paraphrase of the original template, yielding the same semantics when parsed. Any sentences that turn out ungrammatical or have a different meaning indicate an error among the facts used to generate them.

Thus, to recap, the generative story for CNLU starts by sampling a verb semtrans and one or more of its valence patterns, finds role fillers for each valence pattern by asking the user or an LLM for typical phrases that play that role in sentences with the verb, conjoins the fillers into a complete sentence, and parses that sentence to extract a semantic interpretation that can be used for generation. Pseudocode for this process can be seen in Figure 19. Timeouts and validity checks are omitted for brevity.

Note that this process can only generate a subset of the semantics produced by CNLU, but it is sufficient for mining for errors in verb semtranses, valence patterns, and the rules for simple declarative sentences. Extending the generative story to modals, quantifiers, and other complex constructs is a promising area of future work, contingent on improvements to the NLG capabilities of say.

During sentence generation, GEDEM constrains say to reproduce role fillers the same way they were given, so any wrong choices are not exposed to the user. Removing this constraint would allow GEDEM to generate paraphrases of its role fillers as well (e.g., “dish” for “saucer”), potentially exposing more errors, but the fillers would have to be disambiguated correctly to avoid undue noise (e.g., “UFO” for “saucer”). In this work, I focus on verb and valence pattern errors specifically, so I regenerate the fillers verbatim.

```

// Given numeric parameters  $n_s, n_v, n_f$ :
// Return a set of interpretations that can be used to generate examples.
Function SAMPLE-INTERPRETATIONS( $n_s, n_v, n_f$ ):
   $S = \{\}$  // Set of seeds (verb valence patterns).
   $I = \{\}$  // Set of interpretations (semantic expressions).
  // Sample  $n_s$  valence patterns to use as seeds.
  Until  $|S| == n_s$ :
    // Pick a verb semtrans at random from the KB.
     $st = \text{SAMPLE-VERB-SEMTRANS}()$ 
    // Pick up to  $n_v$  valence patterns from the semtrans.
     $S = S \cup \text{SAMPLE-VALENCE-PATTERNS}(st, n_v)$ 
  // Sample interpretations from the seeds.
  For  $vp$  in  $S$ :
    // Elicit  $n_f$  fillers per valence pattern seed.
    For  $i = 1$  to  $n_f$ :
       $f = \{\}$  // One set of role fillers for  $vp$ .
      For  $r$  in  $\text{ROLES}(vp)$ :
        // Query an LLM for a valid phrase to fill role  $r$  in  $vp$ .
         $f = f \cup \text{ELICIT-ROLE-FILLER}(vp, r, f)$ 
      If  $\text{FILLERS-COMPLETE}(vp, f)$ :
        // Construct a sentence from the fillers.
         $s = \text{JOIN-FILLERS}(vp, f)$ 
        // Parse the sentence and extract an interpretation.
         $I = I \cup \{\text{EXTRACT-INTERPRETATION}(s)\}$ 
  Return  $I$ 

```

Figure 19. GEDEM generative story for CNLU.

5.1.2 Generating and Judging Examples

Once the semantics have been sampled, the next stage of GEDEM is to generate a cluster of related sentences that reflect those semantics. If the system is capable of both understanding and generation, the cluster should be a set of sentences that all share the same interpretation when parsed. Any discrepancies in meaning or grammatical errors indicate that there is an error in the linguistic knowledge used to generate those sentences.

Note that the errors found this way are not necessarily related to the linguistic knowledge used in the generative story. For example, the verb semtrans, valence pattern, and role fillers used to sample semantics for CNLU may all be perfectly fine, yet the generated sentences may

still have issues. This can occur when the generated sentences use a different semtrans or valence pattern to convey the same semantics, when a lexicon entry is incorrect, or when one of the sentences is generated using a buggy grammar rule.

Thus, although the generative story only looks at verb semtranses and valence patterns, it can uncover errors in all kinds of linguistic knowledge, depending on how far afield the system is allowed to go while generating the sentences. The advantage of this is that GEDEM does not have to be looking for anything in particular in order to find errors. Just running it on new examples will be enough to uncover new problems to fix, subject to the coverage of the generative story and the constraints on text generation.

The disadvantage of this is that GEDEM can be difficult to steer towards specific errors. For example, testing a particular rule with GEDEM requires sampling semantics that can be expressed with that rule and either forcing the NLG system to use that rule during generation or generating sentences until one of them uses it. Targeting specific semtranses, valence patterns, or lexicon entries is easier because they are closely connected to the semantics that are used to generate the sentence, but forcing the system to try a particular fact may require fine-grained control over generation.

Another consideration when generating examples for GEDEM is how many sentences the system can generate from the same set of semantics. GEDEM gets leverage by comparing one generated sentence to another. If the system cannot generate enough options, it won't have anything to compare. In the extreme case, if only one word maps to a concept, then all generated sentences using that concept will have to use that word, so there will be nothing to check it against. Even in this case, the possibility of buggy lexicon entries or bad grammar rules makes

GEDEM worthwhile, but as a rule, GEDEM works better when it has more options to pick from.³⁰

In general, CNLU has enough redundancy to avoid this issue, but it can crop up in situations where coverage is sparse or a word lacks synonyms. As a coarse measure of redundancy, CNLU has roughly 15,900 semtranses that refer to the same collection as another semtrans and 13,700 that refer to unique collections. For verbs, the focus of current GEDEM efforts, the ratio is more favorable: 5,300 verb semtranses refer to the same collection as another verb semtrans, and 1,900 verb semtranses refer to a unique collection. This suggests that most of the long tail semtranses belong to nouns, and that additional tweaks to generation may be necessary to debug words that cannot be directly paraphrased, such as using clarifying phrases, switching from paraphrase judgments to entailment judgments, or asking about the semtrans directly using an INLD-style question.

The CNLU implementation of GEDEM uses *say* to generate sentences from semantics. For reasons of length, *say* is described separately in Section 5.2. The system then elicits

```
Consider the word "dodges", meaning avoiding (a performing).
The following sentences should mean the same thing as:
    "the clever fox dodges the trap"
Which of them is ungrammatical or means something different?

1) The clever fox skirts the trap
2) The clever fox evades the trap
3) The clever fox shuns the trap
4) The clever fox dodges the trap
5) The clever fox avoids the trap

(Enter a list of numbers between 1 and 5, "none", :help, or :exit.)
> 3
```

Figure 20. Example of a cluster of GEDEM examples for the word “dodges”.

³⁰ One way to deal with this problem is to replace paraphrase judgments with entailment judgments. For example, “He hit a baseball.” entails “He hit a ball.” This is similar to the clarification strings used by verbalize for INLD (Section 2.3), and it has the advantage that Recognizing Textual Entailment (Dagan, Glickman, & Magnini, 2005) is a well-studied task. However, monotonicity can complicate this approach: “He did not hit a baseball.” does not entail “He did not hit a ball.” See Manning’s work on natural logic for a treatment of the issues involved (Angeli & Manning, 2014; MacCartney & Manning, 2007; MacCartney & Manning, 2009).

paraphrase/grammaticality judgments by asking the user multiple choice questions using the same text interface as INLD, as seen in Figure 20. For each example, the system keeps track of the linguistic facts used to generate it. This allows the system to assign blame for incorrect generations and score the facts that are most likely to be responsible, discussed next.

5.1.3 Scoring Errors

The output of GEDEM is a list of linguistic facts that are most likely to be erroneous, given the incorrect generations found in the previous step. GEDEM calculates a *suspicion score* for each linguistic fact used in generating the examples. There are two broad approaches to calculating suspicion scores in the error mining literature: a global score based on the ratio of incorrect sentences to total sentences that a feature appears in, and an iterative score that assigns blame within each sentence to down-weight features that happen to co-occur with the true culprits.

The “parsability” score introduced by van Noord (2004) is representative of the first approach. The parsability of an n -gram over a corpus is the ratio of the number of sentences containing the n -gram that parsed successfully to the total number of sentences containing the n -gram. de Kok and van Noord (2017) reformulate this slightly, yielding the following equation for the suspicion of feature f_i over bag of sentences S :

$$susp(f_i) = \frac{\sum_{s \in S} f_i(s) \cdot error(s)}{\sum_{s \in S} f_i(s)}$$

Here, $f_i(s)$ is how many times feature f_i occurs in sentence s and $error(s)$ is an indicator variable indicating whether s parses correctly. Features can be words, n -grams, or any other aspect of the sentence to which blame may be assigned. The advantage of this approach is that it is a simple way to highlight features that are associated with broken parses.

The second approach, introduced by Sagot and de la Clergerie (2006) and refined by de Kok and van Noord (2017), defines the suspicion score of a feature recursively. The global suspicion score for a feature takes into account not just the number of erroneous sentences a feature appears in, but also its share of the suspicion in each particular sentence, as determined by the local suspicion score for each feature in that sentence. The local suspicion score for a feature in a particular sentence is defined as the ratio of that feature's global suspicion score to the total global suspicion score for features in that sentence.

Starting with a uniform distribution of suspicion within each sentence, the global and local scores can be calculated iteratively until they converge. This yields a global suspicion score that reflects whether a particular feature is truly to blame for the erroneous sentences it appears in or whether its appearances are just incidental.

For the purposes of GEDEM, the first approach is sufficient. Unlike traditional error mining, GEDEM generates clusters of sentences that can be compared to each other while assigning suspicion. Because of the high degree of overlap between generated sentences in a cluster (shared role fillers, words that don't vary, etc.), typically only a few facts will differ between them.

As such, suspicion should only be assigned to facts that were used to generate the incorrect sentences in a cluster but not any of the correct ones. It is difficult to imagine a case where one error can cancel out another, at least when generating with say, so if a fact can be used to generate a correct sentence, it should not be held responsible if it is applied in the same way to generate an incorrect sentence.

The requirement that the facts be used the same way is a subtle but important distinction. In CNLU, lexicon entries and grammar rules have features that can take multiple values. Some

of these values may be correct, and others may be errors. For example, suppose the lexicon entry for the verb “likes” incorrectly lists its agreement as third-person singular *and* third-person plural. That entry could then be used to generate both the phrases “he likes”, which is correct, and “they likes”, which is incorrect. Thus, even though the same lexicon entry is used in both generations, it is used differently; one of its features has a different value. No suspicion should be assigned to the third-person singular usage, but it should be assigned to the third-person plural usage that caused the incorrect generation.

Putting it all together, the GEDEM implementation for CNLU assigns suspicion scores as follows. Let f_i be an *instantiated* linguistic fact: a fact with values assigned to any relevant grammatical features. (E.g., the hypothetical third-person singular and third-person plural versions of “likes” are treated as different instantiated facts, even though they are stored as one lexicon entry in the KB.) Let s be a generated sentence, let C_s be the cluster that contains it, let $Corr(C_s)$ be the set of correct sentences in C_s , and let $F(s)$ be the set of facts used to generate s .

Next, define $Inc(s)$ for a sentence s as the subset of $F(s)$ that was not used to generate any correct sentence in C_s . Note that $Inc(s)$ will be empty if s is correct.

$$Inc(s) = F(s) \setminus \bigcup_{s' \in Corr(C_s)} F(s')$$

Then the suspicion score $susp(f_i)$ for instantiated fact f_i is:

$$susp(f_i) = \frac{\sum_{s \in S} \mathbf{1}[f_i \in Inc(s)]}{\sum_{s \in S} \mathbf{1}[f_i \in F(s)]}$$

In other words, suspicion is assigned to an instantiated fact whenever it is used to generate an incorrect sentence but not any of the correct sentences in the same cluster. Using this score, GEDEM returns a ranked list of facts believed to be erroneous.

5.2 Say

Say is a sentence-level NLG module within CNLU. Given a semantic expression of the type produced by CNLU, *say* generates English phrases or sentences that convey those semantics. Specifically, *say* tries to achieve parity with CNLU's semantic parsing: Any text it generates will produce the same interpretation it was given when parsed by CNLU, if the right choices are selected. Likewise, if CNLU parses a sentence, *say* should be able to regenerate it, an example of round-trip generation (Goodman & Bond, 2009). An example of *say*'s input and output can be seen in Figure 17: Given the semantics on the left, it will produce sentences like the ones on the right.

Aiming for parity between understanding and generation serves two main purposes. First, it allows CNLU's linguistic knowledge to be reused for generation, in contrast to NLG systems like *verbalize* that maintain their own, separate body of linguistic knowledge purely for generation. Improvements to understanding should lead to improvements in generation, and vice versa.

Second, aiming for parity opens up CNLU to debugging techniques like GEDEM, which rely on generation to reveal errors. While *verbalize* can express parts of an interpretation well enough for INLD, the ability to paraphrase an entire interpretation at once, as well as generate using the same knowledge used for understanding, streamlines the process of finding errors at scale. It is also crucial for finding errors of permission, which are rarely obvious when looking at NLU alone.

In practice, achieving parity poses several challenges. CNLU, its parsing algorithm, and its grammar were not designed with generation in mind, so care must be taken to capture the same space of possibilities without introducing discrepancies. Likewise, *say* should avoid

introducing generation-specific linguistic knowledge as much as possible, both to reduce the maintenance burden and to ensure that understanding and generation can be debugged in tandem. (For future applications, where fluency is important, it may be worthwhile to apply generation-specific heuristics or map new generation problems to saved, known-good examples.)

The basic approach I have taken in designing say is to reuse as much of the semantic parser as possible. Reversing the parsing algorithm is a daunting prospect given the complexity of the code responsible for constructing the `sem` field of new constituents, which handles grammatical function substitutions, valence pattern expansions, type checking, choice set maintenance, merging nested conjunctions, converting quantifiers into a canonical form, and processing the collection placeholders used by adjective and adverb semtranses. In order to achieve parity with CNLU's parser, a generation algorithm must not only replicate all of this functionality but be kept up to date as the parsing algorithm changes.

To sidestep this problem, say takes a *generation by parsing* approach, where the same code used for parsing drives the generation algorithm. The agenda is initialized with constituents that could provide pieces of the desired interpretation, then the parsing algorithm is run to combine them into parse trees that yield said interpretation, from which the output text can be decoded. In essence, say parses a whole space of potential sentences at once, keeping only the ones that produce the input semantics.

In the remainder of this section, I walk through each of these steps in turn. Section 5.2.1 discusses the problem setup, namely how the agenda is initialized and what changes need to be made to the chart. Section 5.2.2 lays out how the CNLU parsing algorithm can be used to generate parse trees for the given semantics. Section 5.2.3 discusses how say decodes these parse trees into text. Section 5.2.4 touches on some of the constraints that can be used to control

say and guide its output. Finally, Section 5.2.5 weighs the limitations of this approach and how much parity with understanding can currently be achieved in practice.

5.2.1 *Setup*

To produce parse trees, CNLU's parsing algorithm needs something to parse. For NLU, the input is a sequence of tokens, which are converted into constituents corresponding to their lexicon entries. These constituents are then added to the agenda and inserted into the chart one at a time. Any sequences of constituents that match the right-hand side of a grammar rule will create new constituents to add to the agenda, causing the cycle to continue.

For NLG, the situation is not as straightforward. The output is still a set of parse trees that capture the semantics of the input, but in this case, the semantics are known in advance and the words and their ordering are variable. Thus, there are three aspects of the problem that must be accounted for: covering the desired semantics, accounting for word variability, and dealing with the unknown word ordering. I address each of these in turn.

To produce a parse tree with the right semantics, say needs the right building blocks. It begins by walking through the input semantics, carving them up into individual statements, and retrieving all semtrances that could cover at least one of those statements. For example, if the input semantics contain `(isa eat4200 EatingEvent)`, say will retrieve all semtrances that could produce this expression with the correct grammatical function binding. There are six such semtrances in the KB, corresponding to the verbs “eat”, “gobble”, “munch”, “snack”, “sup”, and “tuck”. Say also keeps track of what grammatical function bindings these semtrances need to produce the target semantics. In this case, all six semtrances have semantics of `(isa :ACTION EatingEvent)`, so `:ACTION` must be bound to `eat4200`.

Say now has all the semtranses it needs to produce a parse tree with the desired semantics, assuming the request is valid and one can be found at all. The next step is to combine these semtranses into constituents to initialize the agenda. Say groups the semtranses according to their part of speech, semantics, head variable, bindings, and the grammatical features of their corresponding lexicon entries. If two or more semtranses share all of these attributes, they can safely be grouped into a single constituent with that part of speech, that head variable, and those grammatical features. They are effectively equivalent, and the normal parsing process can handle any remaining discrepancies, such as differing valence patterns. If any of the attributes differ, the semtranses will be assigned to different constituents.

This approach is how say handles variability in word choices. All possible word choices for the target semantics are added to the agenda for consideration, and some of them are grouped together into a single constituent for compactness. In these constituents, the semtranses are added as a choice set in the `sem` field, the same way one word with multiple meanings would be handled in normal parsing. Their corresponding tokens are stored as a parser variable in the `lex` feature. Each of these constituents represents a set of equivalent word choices in the same position. The `var` field is the discourse variable that needs to be applied to the semtranses to yield the desired semantics.

Those constituents suffice to give say the semantics it needs, but by themselves, they are not enough to generate complete, grammatical sentences. Say also must account for *function words*, such as determiners, prepositions, pronouns, or auxiliary verbs, which help define the grammatical structure of the sentence. Not all of these words have semtranses in CNLU, so they are not instantiated by semtrans lookup with the rest of the *content words* (nouns, verbs, etc.) and must be handled separately. To keep the number of constituents manageable, say merges all the

function words with a particular part of speech into one constituent, again using a parser variable in the `lex` feature to capture the space of options. These placeholder constituents are added to the agenda, allowing say to fill in function words as needed when matching rules.

The last issue that must be taken into account is unknown word ordering. In normal parsing, every constituent has a fixed position in the chart, grounded in the sequence of input tokens. But say does not know the order of its words in advance; that is part of what it is trying to determine. Any word under consideration can appear in any position, and say will not know which of those positions are valid until it has completed a parse tree.

This leads to an insight. Because any word can appear in any position, all positions are equivalent. Any constituent added to the chart in one position should be added in all other positions as well because there is always an arrangement of words that would lead to its being there. This means that say only needs to keep track of one position in its chart; all other positions are exact copies of it and do not need to be considered explicitly. Instead, say treats its chart as *circular*: Sequences of constituents are found by walking over the same position repeatedly until all possible rule matches have been found.

Walking a circular chart efficiently requires a slight tweak to the traversal part of the parsing algorithm. Given a new constituent, say looks up all rules looking for a constituent of that type. Then for each rule, say searches for all sequences of constituents that could match the rule's right-hand side, only considering those that contain the new constituent. The rule is applied to these sequences as normal, and any new constituents created from these matches are added to the agenda so they can be processed later. In this way, say considers all possible word orderings without having to store redundant information. It also reuses as much of the CNLU

parsing algorithm as possible, avoiding discrepancies and paving the way for parity between understanding and generation.

5.2.2 *Generating Parse Trees*

With the agenda populated and a circular chart instantiated, say is ready to begin parsing. Modulo the traversal changes, the core algorithm is the same as CNLU: pop a constituent off the agenda, add it to the chart, fire any rules it matches, and add the resulting constituents to the agenda. Eventually, this will yield a constituent of the desired type which covers all of the target semantics, constituting a complete parse tree. In essence, the parse tree is a proof that any of the sequence of tokens encoded by the tree's leaves will yield the target semantics when parsed by CNLU. There are a few more cases that need to be considered before generating the final output, but once a suitable parse tree has been found, the bulk of the work has been done.

To keep parsing on track, say implements several filters. One is *semantic subsumption*: Any constituent whose semantics do not *subsume* the target semantics is dropped from consideration. This deals with the various ways a constituent's semantics could move in an undesired direction, such as rules adding extra semantic expressions, semantic expressions having the wrong quantifier scopes, or valence patterns producing role relations that are not in the target. The setup step gave say all the pieces it needs; semantic subsumption checks ensure that they are assembled in a valid way.

Semantic subsumption is implemented as a function that walks over a constituent's `sem` field and compares it to the target semantics, failing if the constituent contains any expression or quantifier that is incompatible with the target. The function takes into account conjunctions, choice sets, unbound grammatical functions, and other factors that affect the subsumption relationship. The same function is used to identify complete parse trees: When the parser creates

a constituent of the desired phrase type whose semantics both subsumes and is subsumed by the target semantics, it is logged as a candidate for generation.

Another one of say's filters is a check for redundancy. This is necessary to avoid the same constituent being used in multiple places in the same parse tree. When say instantiates its initial constituents from the semtranses it retrieves, it notes which ones cover the same piece of semantics. From then on, those constituents cannot be merged into the same tree, so each piece of the target semantics will only be covered once.³¹ There are no redundancy constraints on the constituents for function words, so say is free to use as many function words as it needs to complete the sentence.

Finally, say does not allow constituents to merge if they require different bindings. When the constituents are instantiated, the algorithm records what grammatical function bindings are needed to produce the desired semantics. If two constituents require different bindings for the same grammatical function, there is no point in trying to combine them, as any binding of that grammatical function that yields the right semantics for one will yield the wrong semantics for the other. As new constituents are created, say keeps track of which open bindings they still need to produce the right semantics and which ones have been satisfied.

5.2.3 *Decoding Output*

Once the parsing algorithm has produced one or more parse trees that cover the target semantics, the final step is *decoding* them into output strings. Each parse tree found by say

³¹ Multiple appearances of the same semantics in the same sentence can be handled, but they require different discourse variables so they are not collapsed into the same expression. That is, the input to say should reflect the way CNLU parses a sentence: by giving each *word* its own discourse variable. So to generate a sentence like “The man put on the man’s coat”, say would require both instances of “man” to have separate discourse variables. It is up to the caller of say to process the input semantics into this format. Fleshing out this step is an area for future improvements.

```

// Given semantics s, parsing constraints Cn, time limit t:
// Return a set of sentences that CNLU could parse as s.
Function GENERATE(s, Cn, t):
  A = MAKE-AGENDA() // The agenda, a priority queue.
  Ch = MAKE-CIRCULAR-CHART() // The chart, an array.
  G = RETRIEVE-GRAMMAR() // The grammar, a set of rules.
  P = MAKE-QUEUE() // Queue of parse trees found so far.
  O = {} // Set of output strings.
  // List of constituents for semtranses that cover part of s.
  For c in CONSTITUENTS-FOR-CONTENT-WORDS(s):
    ENQUEUE(A, c)
  // List of constituents containing placeholder function words.
  For c in CONSTITUENTS-FOR-FUNCTION-WORDS():
    ENQUEUE(A, c)
  // Generate until out of constituents to process or timeout.
  Until EMPTY(A) or TIME-ELAPSED() > t:
    If EMPTY(P):
      // Search for more parse trees.
      c = DEQUEUE(A)
      If CHECK-CONSTRAINTS(c, Cn):
        // Only allow constituents with compatible semantics.
        If SEM-SUBSUMES(SEMANTICS(c), s):
          INSERT(Ch, c)
          // c is a solution if it covers all of the target semantics.
          If SEM-SUBSUMES(s, SEMANTICS(c)):
            ENQUEUE(P, c)
          // Check every rule that could contain c.
          For r in ALL-APPLICABLE-RULES(G, c):
            // Walk the chart for constituent sequences with c.
            For seq in ALL-APPLICABLE-SEQUENCES(Ch, r, c):
              // Try to apply r to seq to make a new constituent.
              cnew = APPLY-RULE(r, seq)
              If cnew:
                // Add the new constituent to the agenda.
                ENQUEUE(A, cnew)
    Else:
      // Decode the parse trees into string outputs.
      pt = DEQUEUE(P)
      // Ensure decoded outputs have same syntactic structure.
      con = MAKE-TREE-CONSTRAINT(pt)
      // Decode by parsing again, this time without merging words.
      O = O ∪ DECODE(s, pt, con)
  Return O

```

Figure 21. The say generation algorithm.

represents a space of possible sentences that share the same syntactic and semantic structure, as

encoded by the lists of tokens in the `lex` feature of each leaf constituent. There are several possible ways to perform decoding, but `say` uses a second round of parsing, this time constrained by the parse tree found during the first. The process is largely the same, but instead of bundling multiple words into one constituent, `say` adds the constituents normally and allows the parsing algorithm to build them up into several competing parse trees. Each of these new parse trees represents a different, *fixed* sequence of words that conforms to the parse tree found during the previous step. Any constituent that does not fit the tree can be immediately discarded, so this step is fast. Finally, `say` reads the tokens off the leaves of the new parse trees and *detokenizes* them into one output string per tree.³² Strings can be mined until the options are exhausted, or until a set limit is reached, at which point `say` will continue searching for a new parse tree to decode. Putting this all together, pseudocode for `say` can be seen in Figure 21.

It is worth noting that `say` does not exercise direct control over the decoding process. Whichever combination of word choices the parsing algorithm explores first will be the first string returned. If more control is needed, decoding can be guided by parsing constraints, described next.

5.2.4 Constraints

As alluded to above, CNLU has a mechanism for constrained parsing, which drops any constituents that violate a constraint. The constraints are modular and can be configured to meet the needs of specific parsing problems. For example, constraints can enforce span boundaries and phrase types, allowing CNLU to receive advice from an expert user or an external syntactic parser. They can exclude specified rules, reducing the search space by preventing undesired

³² Detokenization also reverses CNLU's internal bookkeeping for punctuation and the like, producing a string of English text of the kind that CNLU would accept as input.

parses. Or they can require CNLU to produce a parse tree of a specified form. This last type of constraint is how say decoding works: Any constituent that is incompatible with the parse tree found in the previous step is filtered out. Constraints are the easiest way to guide the output of say in cases when something is already known about the desired output.

The CNLU implementation of GEDEM uses constraints to ensure that the phrasing of role fillers is that same as that provided by the user or LLM when examples are generated. Since the role fillers are meant to make the example concrete, enabling other linguistic knowledge to be tested, keeping them fixed reduces variability and avoids the need for correct disambiguation. Other constraints could also be used to force say to explore particular words or phrasings, but this requires a more purposeful approach to testing than the current GEDEM implementation has.

More generally, constraints open up a wide toolkit for guiding say, such as using certain phrase types, enforcing specific word orders, or generating phrases that match previously seen examples. As say is deployed to other applications beyond debugging, giving it the right set of constraints will help it produce more fluent text with less work.

5.2.5 *Limitations*

Say still suffers from two drawbacks: CNLU’s grammar was not designed for generation, and the size of the search space can quickly become intractable. In practice, the former is simply a grammar engineering issue: There are quite a few rules that are infinitely recursive, overly permissive, or otherwise buggy when applied to generation. Fixing them is necessary to bring CNLU’s NLG capabilities up to the same level as its NLU, so I do not change the say algorithm to try to accommodate them. Instead, they will be fixed as they become relevant.

The latter problem is more nuanced. Because say uses placeholders for function words, and because there are hundreds of grammar rules that handle different sentence structures, the

number of constituents to explore grows rapidly. To guide the search towards more likely candidates, say uses the following hand-tuned scoring heuristic:

$$Score(c) = 1.4 * SemScore(c) - 0.2 * TreeScore(c) - 0.1 * BindingScore(c)$$

Here, $SemScore(c)$ is the number of atomic semantic expressions in c 's semantics, a measure of semantic coverage; $TreeScore(c)$ is the number of constituents in the parse tree rooted at c , which penalizes overly complicated syntax; and $BindingScore(c)$ is the number of open grammatical functions in c 's semantics, which encourages say to bind them (e.g., assign subjects and objects) rather than leaving them open. Additionally, leaf constituents are given the large, arbitrary score of 150 to ensure that a crucial word is not left trapped on the agenda while say explores an early option with a high score.

The scoring heuristic helps, but it is not enough to solve the tractability issue by itself. There are a number of other options that may improve search, such as training a machine learning model to score constituents, using analogy to guide the search by adapting previously seen examples, or introducing commonsense constraints to guide generation towards common, straightforward linguistic constructs and away from the more esoteric parts of the grammar.

However, for the purposes of GEDEM, improving search is a double-edged sword. GEDEM aims to reveal linguistic errors through generation; driving generation towards known-good parts of the search space obscures errors in knowledge rather than exposing them. Thus, to reduce say's search space without obscuring errors and to avoid tripping over rules that are known to be bad for generation, I use a whitelist of 42 rules, roughly 9% of CNLU's complete set of 467 rules. Grammar rules not on this list are ignored by say. The list contains rules that handle core functionality like binding verb subjects and objects, as well as rules for promoting

verb phrases into complete sentences. These rules are some of the most frequently used for parsing, and they are a good fit for generation, although they are not necessarily error-free.

Using a whitelist decreases the odds of incidentally finding an erroneous rule with GEDEM because it filters out the worst offenders ahead of time. However, the improved tractability and focus on a smaller set of rules makes it easier to find errors in semtranses and valence patterns. As grammar rules are modified to support generation, they can be added to the whitelist to improve the diversity of say’s output and the breadth of grammatical constructs the CNLU implementation of GEDEM can cover.

5.3 Evaluation

I evaluate my implementation of GEDEM by sampling a set of valence patterns from the KB, eliciting role fillers from an LLM according to the process described in Section 5.1.1, and generating clusters of examples for two humans and one LLM to judge. Specifically, I sample verb semtranses at random and select up to 3 valence patterns per semtrans until I have 100 valence patterns to use as seeds. Next, I elicit role fillers from phi4 (Abdin, et al., 2024), a 14B parameter LLM from Microsoft running on Ollama.³³ Each valid set of role fillers is used to construct one interpretation for its valence pattern. For each interpretation, say tries to generate a cluster of up to 5 examples. It may generate less if it runs out of options to try (due to limited synonyms for the verb and/or the limited space imposed by the whitelist) or if it reaches a 30-second timeout. These clusters are then presented to the user or LLM for judgment, as shown in Figure 20. Finally, sentences marked as bad paraphrases or ungrammatical are used to calculate suspicion scores for the facts used to generate them. This pipeline is summarized in Figure 22.

³³ <https://ollama.com/library/phi4:14b>

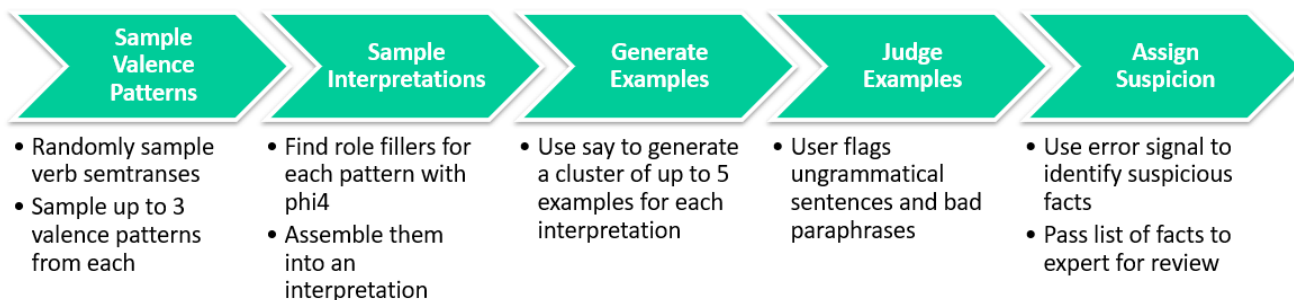


Figure 22. The GEDEM pipeline.

This experiment is a revision of the one reported by Nakos, Demel, and Forbus (2025). The primary changes are upgrades to say that improve its performance and expand the types of sentences it can cover, the new role filler elicitation method discussed above, and a switch to phi4 for the LLM used to find role fillers and perform judgments.

Beginning with 100 valence patterns as seeds, GEDEM queried phi4 for up to 3 sets of role fillers for each pattern. The query was repeated until a full number of sets were found that passed the check, or until a 60-second timeout was reached. Pre-filtering the fillers slows down this process compared to the previous iteration of this experiment, but it drastically improves the number of usable sets of fillers GEDEM is able to find.

The system was able to find 258 total sets of role fillers covering 86 out of 100 seed valence patterns. It then attempted to convert these sets of fillers into interpretations. It was able to produce 128 interpretations covering 52 of the original seed valence patterns. The remaining 130 sets of role fillers failed to validate.³⁴ The system then applied say to these interpretations to generate 209 examples for 53 interpretations. 75 interpretations failed to generate any examples, 42 due to a 60-second timeout and 33 because they exhausted all possible constituents that fit the constraints of the problem and the rule whitelist.

³⁴ This is a second set of checks that test the suitability of the complete interpretation, rather than the checks used to test individual fillers during elicitation.

The end result was 209 examples grouped into 53 paraphrase clusters. Out of these, 26 clusters found all 5 requested examples, 3 clusters found only 4, 19 clusters found only 3, and 5 clusters found only 2. There were no singleton clusters; every example that was generated had at least one paraphrase to compare against.

The attrition rate of this process remains high, largely due to imperfect coverage for say. However, the throughput of the algorithm shows a marked improvement from the previous version of the experiment, which took 200 seeds to find 113 interpretations and generate only 24 clusters of examples. As say approaches parity with CNLU, the throughput of the algorithm will increase (at the cost of an increased search space during generation). Moreover, the role fillers, interpretations, and example generation attempts that fail along the way are a rich vein of examples to debug CNLU and say (e.g., “the paper the paper cuts”, a set of role fillers that fails validation due to the degenerate double-subject valence pattern it was elicited from).

Once the examples were generated, two annotators and phi4 answered 53 questions, marking any examples that were ungrammatical or had mismatched meaning. All participants marked similar numbers of examples wrong: 118 right vs. 91 wrong and 114 right vs. 95 wrong for the annotators, and 121 right vs. 88 wrong for phi4. This confirms that say does, in fact, generate buggy examples, although the extent to which it does so depends on what rules are in play and what systematic errors need to be fixed.

Agreement between the two annotators was high: Fleiss’s $\kappa = 0.79$ and the raw agreement was 89%. This indicates that the task is well-formed and, with appropriate instructions, human annotators can perform it consistently. phi4 showed moderately strong agreement with both annotators: $\kappa = 0.56$ and a raw agreement of 78% for the first, and $\kappa = 0.60$ and a raw agreement of 80% for the second. While not a perfect match for human performance, the LLM does well

enough that the prospect of full automation is reasonable. Notably, GEDEM’s statistical nature means that, over time, any errors in judgment the LLM makes are likely to come out in the wash. With improved say coverage and a smarter sampling procedure to follow up on suspicious facts, GEDEM with an LLM oracle could hunt for errors indefinitely.

As far as finding errors is concerned, the experiment was a mixed success. The system showed similar patterns of behavior as the initial version of the experiment reported in Nakos, Demel, and Forbus (2025).

```
Consider the word "stigmatizes", meaning evaluating goodness (an evaluating).
The following sentences should mean the same thing as:
    "The media stigmatizes poverty"
Which of them is ungrammatical or means something different?

1) The media esteems poverty
2) The media prizes poverty
3) The media values poverty
4) The media deplores poverty
5) The media admires poverty

(Enter a list of numbers between 1 and 5, "none", :help, or :exit.)
> 1 2 3 5|
```

Figure 23. Example of a cluster of GEDEM examples for the word “stigmatizes”.

First and foremost, GEDEM succeeds in the use case that inspired it: exposing erroneous synonyms. Figure 20 shows a bad paraphrase of “dodges” as “shuns”, while Figure 23 shows four bad paraphrases of “stigmatizes” due to an erroneous semtrans. These synonym examples are not the dominant type of cluster that GEDEM generates, but they appear commonly in the test set and serve to highlight a class of error in CNLU’s linguistic knowledge that has never before been explored systematically.

The second noteworthy pattern is the prevalence of bad passives among the generated examples, as seen in Figure 24. CNLU does not keep track of the difference between valence patterns used for active sentences and ones used for passive sentences, so it assigns the active “loud music enrages a baby” and the malformed passive “loud music *is* enraged a baby” the

same semantics. For parsing, the problem is negligible: CNLU will occasionally produce the wrong semantics for a sentence alongside its other interpretations, or it will accept a sentence that is ungrammatical. But for generation, the problem is significant. Imperatives also showed similar issues, with :OBJECTs appearing as grammatical subjects in generated sentences. Fixing these problems will require both reworking the relevant grammar rules and finding and correcting the underspecified valence patterns throughout the KB.

Another issue that crops up is how many paraphrases say can actually generate. While clean examples like Figure 23 show the ideal case, where a word has several synonyms that can be used (even if they are wrong), examples like Figure 24 are also frequent, where a full set of examples cannot be generated and the ones that can follow routine patterns. The former is due to semantic variety: semtranses that map multiple words to the same concept. The latter is due to grammatical variety: multiple rules that can be used to express the same types of semantics. The former is limited by synonym coverage in the KB, while the latter is limited by the rules allowed by the whitelist and, in the case of timeouts, how well say can sift through them.

```
Consider the word "enrages", meaning causing raging (an event).

The following sentences should mean the same thing as:
    "loud music enrages a baby"
Which of them is ungrammatical or means something different?

1) Loud music enrages a baby
2) Loud music has enraged a baby
3) Loud music is enraged a baby

(Enter a list of numbers between 1 and 3, "none", :help, or :exit.)
> 3
```

Figure 24. Example of a cluster of GEDEM examples for the word “enrages”.

While the current setup is sufficient to reveal verb and valence pattern errors, it will need to be expanded to go much farther. More complex interpretations, a wider range of rules to vary up the syntax, and the ability to paraphrase noun phrases will all open up new types of sentences

to debug. More work may be needed to test words without synonyms, perhaps by generating entailments using related concepts instead of exact paraphrases.

Finally, suspicion scores themselves are not as directly useful as hoped. The scores tend to cluster around the extremes. Out of 656 facts that were assigned scores in this experiment, 199 of them had the maximum suspicion score of 1 and 384 had the minimum suspicion score of 0. This made the ranked list of facts difficult to browse, especially when those facts were taken in isolation. Switching to an iterative scoring method like de Kok & van Noord (2017) might help assign blame more accurately, but it is likely that the system needs a larger dataset to pin down the culprits. Targeted sampling is one way to achieve this: Rather than waiting for random chance to gather more data about suspicious facts, the system should specifically search for examples that test them. This approach is well-suited to an LLM oracle, where evaluation and sampling can be more easily interleaved than with a human annotator.

However, although the suspicion scores themselves are not yet useful, the record of what facts were implicated in which bad examples is a valuable debugging tool. True to the spirit of GEDEM, concrete examples are easier to spot errors in than abstract ones, and every suspect fact comes with a set of good and bad example sentences for comparison. This provides a solid basis for manual debugging, and it could serve as input to an INLD system set up to compare multiple sentences.

In sum, GEDEM is able to generate example sentences that reveal errors of permission in the underlying NL system, subject to the limitations of sampling and generation. Although the proposed suspicion score did not prove immediately useful, the annotated examples and the facts used to generate them provide grist for other forms of debugging. Meanwhile, the experiment revealed several types of errors in CNLU’s linguistic knowledge and say’s coverage: false verb

synonyms, systematic issues with passives and imperatives, and various failed instances of role fillers, interpretations, or examples uncovered during test set generation. Taken with INLD, GEDEM is a step towards easing the maintenance burden of hand-curated NL systems.

6. Related Work

While I am unaware of any prior research on interactively debugging or mining for errors in a semantic parser, I have drawn inspiration from several veins of related work, including model-based diagnosis, Interactive Task Learning, explainability, debugging machine learning systems, error mining for natural language systems, and knowledge engineering techniques for extending and debugging knowledge bases. I discuss each of these in turn.

6.1 Model-Based Diagnosis

Past work on *model-based diagnosis* has shown how a model of an artifact can be used to diagnose faults in domains ranging from physical devices to education. de Kleer and Williams (1987) present GDE and demonstrate its application to digital circuits. Collins (1994) extends traditional model-based diagnosis with fault models generated by a process-centered domain theory, allowing the system to handle novel faults not enumerated by the designer. de Koning et al. (2000) diagnose student errors by comparing their responses to a model reasoning trace of the same question, using hierarchical decomposition to compactly represent a large search space.

I extend this work to a new domain, applying the GDE algorithm to symptom identification and error localization for hand-curated NL systems within the INLD framework. My system also makes extensive use of hierarchical decomposition to keep the search for errors tractable.

6.2 Interactive Task Learning

Interactive Task Learning (ITL; Laird, et al., 2017) deals with systems that learn tasks from explicit instruction. While the modalities and capabilities of these systems vary, they share

a basic pattern of receiving step-by-step instruction from users and asking follow-up questions to iteratively refine the system’s understanding of a desired task. PLOW (Allen, et al., 2007) allows users to teach a software agent how to perform tasks on websites. SUGILITE (Li, Azaria, & Myers, 2017) and PUMICE (Li, et al., 2019) learn to perform tasks within smartphone apps from user demonstration, then compose them into more complicated behaviors. Game-learning systems like Rosie (Kirk & Laird, Interactive task learning for simple games, 2014) and the Companion-based system presented by Hinrichs and Forbus (2014) let users teach the system how to play new games. Wang et al.’s (2017) work on the drawing game VoxelRun lets players compose primitive drawing commands into more complex concepts.

INLD shares ITL’s goal of learning through user interaction, but it focuses on repairing and extending linguistic knowledge rather than teaching the system new tasks. Although several ITL systems, including Rosie, PUMICE, and the VoxelRun assistant, support learning new vocabulary terms, their language learning capabilities are limited in complexity, lack the ability to correct errors, and are centered on tasks for their respective domains. Unifying the broad-coverage language learning and model-based diagnosis techniques of INLD with the grounded task learning of other ITL systems is a promising area for future work.

6.3 Explainability

Explainability is the principle of having computer systems that can be understood by their developers and users. While the opacity of a system may come from the inherent complexity of its inner workings (e.g., the billions of parameters that constitute a neural network) or the expertise required to understand them (e.g., grammar rules in a semantic parser), in both cases,

explainability seeks to bridge the gap between the full complexity of the system and the user’s understanding of it.

I draw particular inspiration from the work of Tintarev and Masthoff (2007), who discuss explanations in the context of recommender systems. They discuss seven possible aims for explanations. These include *transparency*, the ability of the user to understand the system’s inner workings, and *scrutability*, the ability of the user to correct the system when it is wrong, both of which are core goals of INLD.

Other work in *explainable AI* (Adadi & Berrada, 2018) seeks to help users gain insights into AI systems. The bulk of this work focuses on dealing with the complexity of machine learning-based systems, whether by training a classifier to shed light on the system’s local behavior (Ribeiro, Singh, & Guestrin, 2016), determining the importance of features to a decision (Lundberg & Lee, 2017), or training a more explainable model to replicate the behavior of an opaque one (Lindsey, et al., 2025). Hand-curated NL systems do not require these specific techniques, apart from any ML models they use as components, but they share a need for transparency, especially when trying to expand the pool of users who can extend and maintain them.

6.4 Debugging ML Systems

Another line of research involves debugging machine learning systems when they exhibit erroneous behavior. While gathering additional data and retraining the model from scratch remains a common solution, popular techniques such as fine-tuning (Bommasani, et al., 2021) and reinforcement learning with human feedback (Bai, et al., 2022) allow developers to fix the behavior of a pre-trained model when necessary. These techniques share INLD’s goal of

debugging an existing system without recreating it from scratch and GEDEM’s goal of addressing errors at scale, but they fit within the machine learning paradigm of adding data and tweaking model weights rather than building up a base of explicit, verifiable knowledge. Thus, care must be taken that new training does not degrade performance on old tasks (Kirkpatrick, et al., 2017; Niu, Liu, Wang, & Song, 2021) or introduce new errors (Ribeiro & Lundberg, 2022), while newer models must be trained on the updated data as well to avoid falling into the same traps as their predecessors.

One line of research aims to make debugging ML-based NL systems more systematic. Ribeiro et al. (2020) introduces CheckList, a tool for finding and correcting errors in an NL system by systematically varying inputs within certain categories. For example, a developer can test a sentiment classification system by changing the demonyms used in the examples, detecting performance deficiencies or bias in the system. These examples can then be used as additional training data to help fix the system. Ribeiro and Lundberg (2022) expands on this concept by using LLMs to bootstrap examples from a user-provided seed, reducing the amount of effort needed to find and correct errors. Their technique also loops until the behavior of the NL system stabilizes, to prevent the new training data from introducing more bugs.

Likewise, the fields of *machine teaching* (Simard, et al., 2017) and *interactive machine learning* (Chen, et al., 2018) seek to change the relationship between a model and its developer. They opt for smaller and more inspectable models that can be retrained repeatedly, allowing the developer to take a direct role in development and help the model notice patterns it might not have on its own. This approach is analogous to INLD: an iterative teaching loop driven by hands-on examples, user expertise, and robust tools for giving feedback to the system. However,

they use the building blocks of statistical ML rather than symbolic AI. They attempt to align models with human concepts and intuitions rather than build up an explicit body of knowledge.

6.5 Error Mining

Error mining aims to identify errors in a parser by running the parser over a text corpus and identifying n -grams (de Kok & van Noord, 2017; van Noord, 2004) or rules (Goodman & Bond, 2009) that are commonly associated with fragmented parses. Various methods exist for calculating the suspicion score for a feature (e.g., the presence of a specific n -gram in the sentence), ranging from a simple parsability ratio (van Noord, 2004) to iterative scoring that assigns blame on a per-sentence level (de Kok & van Noord, 2017; Sagot & de La Clergerie, 2006). Goodman and Bond (2009) use round-trip generation to identify gaps in an NL system's generation capabilities. The output of an error mining system is a ranked list of features that are suspected to be buggy for an expert to investigate.

GEDEM extends error mining to NL systems that are equipped with semantics. Semantic errors are much harder to detect than syntactic ones, as there is no obvious, universal signal when a semantic parser fails to produce the right interpretation. GEDEM sidesteps this issue by generating multiple sentences for the same semantics and having a user compare them against each other and check them for grammaticality. This provides an error signal so that suspicion scores can be calculated for features (in this case, the facts that were used in generating the sentences).

GEDEM can also be seen as extension of early NLG work by Yngve (1961), which sampled sentences from a generative grammar to reveal grammatical errors. GEDEM extends

this approach to modern NL systems and constrains generated sentences to use the same semantics, expanding the range of errors that can be found.

6.6 Knowledge Engineering

While INLD and GEDEM focus on linguistic knowledge, similar systems have been created for knowledge engineering in general. Given the knowledge acquisition bottleneck (Olson & Rueter, 1987), finding ways to input and debug knowledge efficiently is a key problem for reasoning systems that rely on explicit knowledge. In particular, KRAKEN (Matthews, Curtis, Hines, Kahlert, & Miraglia, 2004) and its interactive dialogue component (Witbrock, et al., 2003) allow subject-matter experts to browse and expand the Cyc ontology. Likewise, KITTEN (Shaw & Gaines, 1987) is a knowledge support system that unifies a suite of tools, allowing users with a range of expertise to contribute knowledge to an expert system. Additionally, the system presented by Kalyanpur (2006) helps users locate and repair errors in OWL ontologies. All three of these systems share INLD's goal of making complex knowledge accessible to users and giving them the tools necessary to extend and maintain it.

Finally, Mittal and Moore (1996) expose errors in knowledge bases by automatically generating examples for an expert user to evaluate, along with brief explanations of which concept each example is meant to express. Their intuition is that bugs are easier to spot when they involve concrete instances as opposed to abstract ontological information. GEDEM applies this same intuition to the linguistic domain, sampling semantics and generating concrete examples to expose errors in linguistic knowledge.

7. Conclusion

In this thesis, I have presented two complementary frameworks for debugging hand-curated natural language systems. Interactive Natural Language Debugging finds and corrects errors in an NL system’s linguistic knowledge by parsing a sentence and asking the user a series of diagnostic questions in natural language. In particular, model-based diagnosis techniques are used to cast a parse trace as a reasoning problem that is solved by asking the user targeted questions about the intended interpretation of the sentence. I have presented an implemented INLD system for CNLU, the NLU component of the Companion cognitive architecture, and shown its utility by running it on a sample of sentences from the COPA corpus, measuring inter-annotator agreement, and analyzing the quality of the diagnoses that were produced. This supports my first claim, that errors in the knowledge of a hand-curated NL system can be diagnosed and corrected without requiring specific expert knowledge.

Generative Example-Driven Error Mining complements INLD by searching for likely errors at scale. Rather than asking detailed questions about specific sentences, GEDEM samples semantics, generates clusters of sentences, and asks users to identify any sentences that are ungrammatical or have a different meaning from the rest of their clusters. These judgments are then used as an error signal to identify linguistic facts that were involved in generating incorrect sentences, allowing likely errors to be found at scale without specific expertise. I have presented an implemented GEDEM system for CNLU, using its say module for NLG, and shown its use in revealing errors in both CNLU’s linguistic knowledge and say itself. This supports my second claim, that errors can be found at scale by generating multiple sentences from the same semantics.

These frameworks are complementary. INLD drills down on errors in individual sentences, focusing primarily on errors of restriction that prevent an NLU system from finding the right interpretation. GEDEM operates over sets of sentences, asks simpler questions, and focuses on errors of permission that result in incorrectly generated text. Both frameworks reduce the level of expertise required to maintain hand-curated NL systems, currently one of the largest barriers to their use. It is my hope that expanding the pool of users able to contribute to these systems and reducing the amount of effort required to debug them will help them flourish, producing inspectable, trustworthy NL systems whose knowledge grows over time.

7.1 Future Work

While INLD and GEDEM address some of the core problems of maintaining hand-curated NL systems, they only scratch the surface of debugging techniques available. In this final section, I lay out some of the many avenues for future work.

The 2x2 grid in Table 2 suggests two additional frameworks to fill the gaps left by INLD and GEDEM. The first would be an extension of INLD that incorporates generation to drill down on errors of permission at the sentence level. Once INLD had found an acceptable interpretation of a parse, a new set of decomposition strategies can generate related sentences, ask the user about them, and systematically test the system's linguistic knowledge for errors of permission. Once say has achieved greater parity with parsing, this would be a natural addition to the INLD system for CNLU.

The second would be an extension of GEDEM that mines for errors of restriction by parsing sentences in a corpus, then searching for potential errors. Traditional error mining works for certain types of errors; a fragmented parse is an easy way to tell that parsing has failed. But

semantics are more challenging to debug. Care must be taken to distinguish between errors in disambiguation (i.e., the wrong choices being selected) and errors in knowledge, which both appear the same when generating paraphrases. In general, the goal of such a framework is to find simple questions like GEDEM's paraphrase/grammaticality judgments that indicate the NL system has failed to produce an acceptable interpretation of each sentence.

INLD has several possible extensions that would improve its ability to handle difficult errors. Questions that can get at grammatical aspects of a parse without requiring too much technical expertise from the user would be a welcome addition, though care must be taken not to increase the search space too much.

As another approach, *differential diagnosis* can be used to compare the interpretations for two related sentences and drill down on their discrepancies. For example, if a sentence fragments, the system could simplify it until it parses, then identify the phrase responsible for it fragmenting. Likewise, if the correct interpretation for a word is missing, the system could substitute in a synonym to see if the problem is fixed. These techniques fit naturally into a model-based diagnosis framework and would expand the set of debugging strategies the INLD system has at its disposal.

Correction remains an important but unexplored stage of the INLD pipeline. The breadth of NextKB makes it difficult to have a non-expert user identify the correct KB concepts needed for a new semtrans, and other types of errors have similar challenges. However, these challenges are not insurmountable, especially with a more flexible user interface, scripts for drilling down on the right correction, and more clarifying knowledge that can be used to distinguish concepts in the KB. Overcoming these challenges would take INLD one step closer to a complete loop where non-expert users can help debug the system.

Global verification is another extension to INLD that could make it more reliable. When considering a potential correction to the system’s linguistic knowledge, it would be useful to see how the change affects other sentences, not just the one currently being debugged. With a database of prior examples, proposed changes can be checked against a range of scenarios and refined further if needed. The trade-off, though, is the need to maintain such a database and the extra cost of checking the other examples.

Finally, both INLD and GEDEM are ripe for automation. Preliminary explorations show that modern LLMs are linguistically adept enough to answer both the shallow, paraphrase-and-grammaticality questions that drive GEDEM and the deeper questions of meaning that drive INLD. While I have not tried to optimize the LLM setups used in this thesis, plenty of options exist to push performance higher and begin scaling up. In principle, given enough time and computing power, a reliable LLM oracle can mine out the space of errors handled by a particular INLD or GEDEM system, allowing a small team of developers to maintain a large hand-curated NL system. In addition to increasing the robustness of the system, this would also shift the focus of development to more interesting problems of knowledge representation and grammar engineering, rather than hunting down bugs or worrying about coverage.

These are only a few areas of future work. Others include applying INLD and GEDEM to other hand-curated NL systems, making INLD more robust to user mistakes, and laying out a formal theory of debugging hand-curated NL systems to bring it more in line with fields such as program verification. With the right support for debugging, I believe hand-curated NL systems can realize their full potential and take their place as a robust and easy-to-use part of the AI toolkit.

References

- Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., . . . others. (2024). Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.
- Adadi, A., & Berrada, M. (2018). Peeking inside the black-box: a survey on explainable artificial intelligence (XAI). *IEEE Access*, 6, 52138–52160.
- Allen, J. (1994). *Natural language understanding* (2nd ed.). Redwood, City, CA: Benjamin-Cummings Publishing Co., Inc.
- Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., & Taysom, W. (2007). Plow: A collaborative task learning agent. *AAAI*, 7, pp. 1514–1519.
- Angeli, G., & Manning, C. D. (2014). Naturalli: Natural logic inference for common sense reasoning. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, (pp. 534–545).
- Artstein, R. (2017). Inter-annotator agreement. In *Handbook of linguistic annotation* (pp. 297–313). Springer.
- Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., . . . others. (2022). Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.
- Barbella, D. M. (2016). *Analogy in Learning by Reading*. Ph.D. dissertation, Northwestern University.
- Barbella, D., & Forbus, K. (2013). Analogical word sense disambiguation. *Advances in Cognitive Systems*, 2, 297–315.

- Berant, J., & Liang, P. (2014). Semantic parsing via paraphrasing. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (pp. 1415–1425).
- Bevilacqua, M., Pasini, T., Raganato, A., & Navigli, R. (2021). Recent trends in word sense disambiguation: A survey. *IJCAI*, (pp. 4330–4338).
- Blass, J. A. (2023). *Extracting and applying legal rules from precedent cases*. Ph.D. dissertation, Northwestern University.
- Blass, J., & Forbus, K. (2017). Analogical chaining with natural language instruction for commonsense reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31.
- Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., . . . others. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., . . . others. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.
- Chang, M. D., Wetzel, J., & Forbus, K. D. (2011). Qualitative and Quantitative Reasoning over Physics Textbook Diagrams. *25th International Workshop on Qualitative Reasoning*.
- Chen, N.-C., Suh, J., Verwey, J., Ramos, G., Drucker, S., & Simard, P. (2018). AnchorViz: Facilitating classifier error discovery through interactive semantic data exploration. *Proceedings of the 23rd International Conference on Intelligent User Interfaces*, (pp. 269–280).
- Chen, Z., & Liu, B. (2018). *Lifelong machine learning* (2nd ed.). Morgan & Claypool Publishers.

- Collins, J. W. (1994). *Process-based diagnosis: An approach to understanding novel failures*.
University of Illinois at Urbana-Champaign.
- Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20, 273–297.
- Crouse, M. (2021). *Question-answering with structural analogy*. Ph.D. dissertation,
Northwestern University.
- Crouse, M., McFate, C., & Forbus, K. (2018). Learning to build qualitative scenario models from
natural language. *Proc. 31st Int. Workshop on Qualitative Reasoning (QR'18)*.
- Dagan, I., Glickman, O., & Magnini, B. (2005). The pascal recognising textual entailment
challenge. *Machine learning challenges workshop*, (pp. 177–190).
- de Kleer, J. (1986). An assumption-based TMS. *Artificial intelligence*, 28, 127–162.
- de Kleer, J., & Williams, B. C. (1987). Diagnosing multiple faults. *Artificial intelligence*, 32, 97–
130.
- de Kok, D., & van Noord, G. (2017). Mining for parsing failures. In *From Semantics to
Dialectometry. Festschrift in honor of John Nerbonne*. (p. 19). College Publications.
- de Koning, K., Bredeweg, B., Breuker, J., & Wielinga, B. (2000). Model-based reasoning about
learner behaviour. *Artificial Intelligence*, 117, 173–229.
- DeCoste, D., & Collins, J. W. (1991). *CATMS: An ATMS which avoids label explosions*.
Evanston: Northwestern University.
- Dehghani, M. (2009). *A Cognitive Model of Recognition-Based Moral Decision Making*. Ph.D.
dissertation, Northwestern University.
- Detroja, K., Bhensdadia, C. K., & Bhatt, B. S. (2023). A survey on relation extraction. *Intelligent
Systems with Applications*, 19, 200244.

- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *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)*, (pp. 4171–4186).
- Dušek, O., Novikova, J., & Rieser, V. (2018). Findings of the E2E NLG challenge. *arXiv preprint arXiv:1810.01170*.
- Ebrahimi, J., Rao, A., Lowd, D., & Dou, D. (2017). Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751*.
- Ettinger, A. (2020). What BERT is not: Lessons from a new suite of psycholinguistic diagnostics for language models. *Transactions of the Association for Computational Linguistics*, 8, 34–48.
- Everett, J. O., & Forbus, K. D. (1996). Scaling up logic-based truth maintenance systems via fact garbage collection. *AAAI/IAAI, Vol. 1*, (pp. 614–620).
- Fillmore, C. J., Wooters, C., & Baker, C. F. (2001). Building a large lexical databank which provides deep semantics. *Language, Information and Computation: Proceedings of The 15th Pacific Asia Conference: 1-3 February 2001, Hong Kong*, (pp. 3–26).
- Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76, 378.
- Forbus, K. D. (2019). *Qualitative representations: How people reason and learn about the continuous world*. MIT press.
- Forbus, K. D., & Hinrich, T. (2017). Analogy and relational representations in the companion cognitive architecture. *AI Magazine*, 38, 34–42.

- Forbus, K. D., & Hinrichs, T. (2019). Qualitative Reasoning about Investment Decisions. *Proceedings of the 32nd International Workshop on Qualitative Reasoning (QR 2019). Macao, China.*
- Forbus, K. D., Chang, M., McLure, M., & Usher, M. (2017). The cognitive science of sketch worksheets. *Topics in cognitive science*, 9, 921–942.
- Forbus, K. D., Ferguson, R. W., Lovett, A., & Gentner, D. (2017). Extending SME to handle large-scale cognitive modeling. *Cognitive Science*, 41, 1152–1201.
- Forbus, K. D., Gentner, D., & Law, K. (1995). MAC/FAC: A model of similarity-based retrieval. *Cognitive science*, 19, 141–205.
- Forbus, K. D., Hinrichs, T. R., de Kleer, J., & Usher, J. M. (2010). FIRE: Infrastructure for Experience-Based Systems with Common Sense. *AAAI Fall Symposium: Commonsense Knowledge.*
- Forbus, K., Garnier, B., Tikoff, B., Marko, W., Usher, M., & McLure, M. (2018). Sketch worksheets in STEM classrooms: Two deployments. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32.
- Forbus, K., Usher, J., Lovett, A., Lockwood, K., & Wetzel, J. (2011). CogSketch: Sketch understanding for cognitive science research and for education. *Topics in Cognitive Science*, 3, 648–666.
- Galescu, L., Teng, C. M., Allen, J., & Perera, I. (2018). Cogent: A generic dialogue system shell based on a collaborative problem solving model. *Proceedings of the 19th Annual SIGdial Meeting on Discourse and Dialogue*, (pp. 400–409).
- Goodman, M. W., & Bond, F. (2009). Using generation for grammar analysis and error detection. *Proceedings of the ACL-IJCNLP 2009 conference short papers*, (pp. 109–112).

- Guha, R. V. (1992). *Contexts: a formalization and some applications*. Stanford University.
- Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., . . . others. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Gwet, K. L. (2014). *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, LLC.
- Hancock, W. W. (2025, August). *Qualitative Spatiotemporal Episodic Memory*. Ph.D. dissertation, Northwestern University.
- Hand, D. J., & Yu, K. (2001). Idiot's Bayes—not so stupid after all? *International statistical review*, 69, 385–398.
- Hinrichs, T. R., & Forbus, K. D. (2014). X goes first: Teaching simple games through multimodal interaction. *Advances in Cognitive Systems*, 3, 31–46.
- Hinrichs, T. R., & Forbus, K. D. (2019). Experimentation in a model-based game. *Proceedings of the Seventh Annual Conference on Advances in Cognitive Systems*.
- Hinrichs, T., & Forbus, K. (2012). Learning qualitative models by demonstration. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26, pp. 207–213.
- Hinrichs, T., & Forbus, K. (2015). Qualitative models for strategic planning. *Proceedings of the Third Annual Conference on Advances in Cognitive Systems, Atlanta, May*.
- Hovy, E., Marcus, M., Palmer, M., Ramshaw, L., & Weischedel, R. (2006). Ontonotes: the 90% solution. *Proceedings of the human language technology conference of the NAACL, Companion Volume: Short Papers*, (pp. 57–60).

- Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., . . . others. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43, 1–55.
- Isermann, R. (2005, January). Model-based fault-detection and diagnosis – status and applications. *Annual Reviews in Control*, 29, 71–85. doi:10.1016/j.arcontrol.2004.12.002
- Kalyanpur, A. (2006). *Debugging and repair of OWL ontologies*. College, Park, MD: Department of Computer Science, University of Maryland.
- Kandaswamy, S., Forbus, K., & Gentner, D. (2014). Modeling learning via progressive alignment using interim generalizations. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 36.
- Kirk, J. R., & Laird, J. E. (2014). Interactive task learning for simple games. *Advances in Cognitive Systems*, 3, 13-30.
- Kirk, J. R., Wray, R. E., Lindes, P., & Laird, J. E. (2022). Improving Language Model Prompting in Support of Semi-autonomous Task Learning. *Improving Language Model Prompting in Support of Semi-autonomous Task Learning*. Retrieved from <https://arxiv.org/abs/2209.07636>
- Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., . . . others. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114, 3521–3526.
- Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35, 22199–22213.

- Laird, J. E., Gluck, K., Anderson, J., Forbus, K. D., Jenkins, O. C., Lebiere, C., . . . others. (2017). Interactive task learning. *IEEE Intelligent Systems*, 32, 6–21.
- Lenat, D. B., & Guha, R. V. (1991). The evolution of CycL, the Cyc representation language. *ACM SIGART Bulletin*, 2, 84–87.
- Li, T. J.-J., Azaria, A., & Myers, B. A. (2017). SUGILITE: creating multimodal smartphone automation by demonstration. *Proceedings of the 2017 CHI conference on human factors in computing systems*, (pp. 6038–6049).
- Li, T. J.-J., Radensky, M., Jia, J., Singarajah, K., Mitchell, T. M., & Myers, B. A. (2019). Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. *Proceedings of the 32nd annual ACM symposium on user interface software and technology*, (pp. 577–589).
- Lindsey, J., Gurnee, W., Ameisen, E., Chen, B., Pearce, A., Turner, N. L., . . . Batson, J. (2025). On the Biology of a Large Language Model. *Transformer Circuits Thread*. Retrieved from <https://transformer-circuits.pub/2025/attribution-graphs/biology.html>
- Lovett, A., Tomai, E., Forbus, K., & Usher, J. (2009). Solving geometric analogy problems through two-stage analogical mapping. *Cognitive science*, 33, 1192–1231.
- Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, 30.
- MacCartney, B., & Manning, C. D. (2007). Natural logic for textual inference. *Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing*, (pp. 193–200).
- MacCartney, B., & Manning, C. D. (2009). An extended model of natural logic. *Proceedings of the eight international conference on computational semantics*, (pp. 140–156).

- Marcus, M., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19, 313–330.
- Matthews, G., Curtis, J., Hines, K., Kahlert, R. C., & Miraglia, P. (2004). *KRAKEN - Knowledge Rich Acquisition of Knowledge from Experts Who Are Non-Logicians*. Tech. rep., Cycorp, Austin.
- McFate, C. J. (2018). *An Analogical Account of Argument Structure Construction Acquisition and Application*. Ph.D. dissertation, Northwestern University.
- McFate, C., Forbus, K., & Hinrichs, T. (2014). Using narrative function to extract qualitative information from natural language texts. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28.
- McLure, M. D., Friedman, S. E., & Forbus, K. D. (2010). Learning concepts from sketches via analogical generalization and near-misses. *Proceedings of the annual meeting of the cognitive science society*, 32.
- McShane, M., & Nirenburg, S. (2021). *Linguistics for the Age of AI*. Cambridge, MA: MIT Press.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.
- Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM*, 38, 39–41.
- Mittal, V. O., & Moore, J. D. (1996). Detecting knowledge base inconsistencies using automated generation of text and examples. *Proceedings of the Thirteenth National Conference on*

- Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, 1*, pp. 483–488. Portland.
- Mohan, S., Mininger, A. H., Kirk, J. R., & Laird, J. E. (2012). Acquiring grounded representations of words with situated interactive instruction. *Advances in Cognitive Systems*.
- Nakos, C., & Forbus, K. D. (2024). Interactively Diagnosing Errors in a Semantic Parser. *Proceedings of the Eleventh Annual Conference in Advances in Cognitive Systems*.
- Nakos, C., & Forbus, K. D. (2025, July). Towards knowledge autonomy in the Companion cognitive architecture. *Cognitive Systems Research*, 101378.
doi:10.1016/j.cogsys.2025.101378
- Nakos, C., Demel, W., & Forbus, K. (2025). Generative example-driven error mining. *Twelfth Annual Conference on Advances in Cognitive Systems*.
- Nakos, C., Kuthalam, M., & Forbus, K. D. (2022). A Framework for Interactive Natural Language Debugging. *Proceedings of the Tenth Annual Conference on Advances in Cognitive Systems*.
- Nakos, C., Rabkina, I., & Forbus, K. (2020). An Analogical Account of Reference Resolution. *Advances in Cognitive Systems*.
- Nakos, C., Rabkina, I., Hill, S., & Forbus, K. D. (2020). Corrective processes in modeling reference resolution. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 42.
- Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, (pp. 968–973).

- Niu, S., Liu, Y., Wang, J., & Song, H. (2021). A decade survey of transfer learning (2010–2020). *IEEE Transactions on Artificial Intelligence*, 1, 151–166.
- Olson, J. R., & Rueter, H. H. (1987). Extracting expertise from experts: Methods for knowledge acquisition. *Expert Systems*, 4, 152–168.
- Olson, T. (2025, June). *A Formal Theory of Norms*. Ph.D. dissertation, Northwestern University.
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., . . . others. (2022). Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35, 27730–27744.
- Paritosh, P. K., & Forbus, K. D. (2005). Analysis of strategic knowledge in back of the envelope reasoning. *Proceedings of the National Conference on Artificial Intelligence*, 20, p. 651.
- Parsons, T. (1990). *Events in the Semantics of English*. Cambridge, MA: MIT Press.
- Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, (pp. 1532–1543).
- Plaat, A., van Duijn, M., Van Stein, N., Preuss, M., van der Putten, P., & Batenburg, K. J. (2025). Agentic large language models, a survey. *Journal of Artificial Intelligence Research*, 84.
- Pollard, C., & Sag, I. A. (1994). *Head-driven phrase structure grammar*. University of Chicago Press.
- Rabkina, I. (2020). *Analogical theory of mind: computational model and applications*. Ph.D. dissertation, Northwestern University.
- Rabkina, I., Kantharaju, P., Wilson, J. R., Roberts, M., & Hiatt, L. M. (2022). Evaluation of goal recognition systems on unreliable data and uninspectable agents. *Frontiers in Artificial Intelligence*, 4, 734521.

- Rabkina, I., McFate, C., Forbus, K. D., & Hoyos, C. (2017). Towards a computational analogical theory of mind. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 39.
- Ribeiro, D. N. (2023). *Reasoning and structured explanations in natural language via analogical and neural learning*. Ph.D. dissertation, Northwestern University.
- Ribeiro, D. N., & Forbus, K. (2021). Combining analogy with language models for knowledge extraction. *Proceedings of the Third Conference on Automatic Knowledge Base Construction*. Virtual.
- Ribeiro, D., Hinrichs, T., Crouse, M., Forbus, K., Chang, M., & Witbrock, M. (2019). Predicting state changes in procedural text using analogical question answering. *7th Annual Conference on Advances in Cognitive Systems*.
- Ribeiro, M. T., & Lundberg, S. (2022, May). Adaptive Testing and Debugging of NLP Models. In S. Muresan, P. Nakov, & A. Villavicencio (Ed.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 3253–3267). Dublin: Association for Computational Linguistics.
doi:10.18653/v1/2022.acl-long.230
- Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). " Why should i trust you?" Explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, (pp. 1135–1144).
- Ribeiro, M. T., Wu, T., Guestrin, C., & Singh, S. (2020, July). Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In D. Jurafsky, J. Chai, N. Schluter, & J. Tetreault (Ed.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (pp. 4902–4912). Online: Association for Computational Linguistics. doi:10.18653/v1/2020.acl-main.442

- Roemmele, M., Bejan, C. A., & Gordon, A. S. (2011). Choice of Plausible Alternatives: An Evaluation of Commonsense Causal Reasoning. *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, (pp. 90–95).
- Sagot, B., & de La Clergerie, É. (2006). Error mining in parsing results. *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the ACL-ACL'06*.
- Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., . . . Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 68539–68551.
- Schuler, K. K. (2005). *VerbNet: A broad-coverage, comprehensive verb lexicon*. Ph.D. dissertation, University of Pennsylvania.
- Shaw, M. L., & Gaines, B. R. (1987). KITTEN: Knowledge initiation and transfer tools for experts and novices. *International Journal of Man-Machine Studies*, 27, 251–280.
- Simard, P. Y., Amershi, S., Chickering, D. M., Pelton, A. E., Ghorashi, S., Meek, C., . . . others. (2017). Machine teaching: A new paradigm for building machine learning systems. *arXiv preprint arXiv:1707.06742*.
- Speer, R., Chin, J., & Havasi, C. (2017). Conceptnet 5.5: An open multilingual graph of general knowledge. *Proceedings of the AAAI conference on artificial intelligence*, 31.
- Steedman, M. (2001). *The syntactic process*. Cambridge, MA: MIT Press.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

- Tintarev, N., & Masthoff, J. (2007). Effective explanations of recommendations: user-centered design. *Proceedings of the 2007 ACM conference on Recommender systems*, (pp. 153–156).
- Tomai, E. R. (2009). *A pragmatic approach to computational narrative understanding*. Ph.D. dissertation, Northwestern University.
- Tomai, E., & Forbus, K. D. (2009). EA NLU: Practical Language Understanding for Cognitive Modeling. *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*. Sanibel.
- van Noord, G. (2004). Error mining for wide-coverage grammar engineering. *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, (pp. 446–453).
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention is all you need. *Advances in neural Information processing systems*, 30.
- Vrandečić, D., & Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57, 78–85.
- Wang, S. I., Ginn, S., Liang, P., & Manning, C. D. (2017, July). Naturalizing a Programming Language via Interactive Learning. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (pp. 929–938).
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., . . . others. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35, 24824–24837.

Wilson, J. R., Chen, K., Crouse, M., Nakos, C., Ribeiro, D. N., Rabkina, I., & Forbus, K. D.

(2019). Analogical question answering in a multimodal information kiosk. *Proceedings of the Seventh Annual Conference on Advances in Cognitive Systems*. Cambridge.

Witbrock, M., Baxter, D., Curtis, J., Schneider, D., Kahlert, R., Miraglia, P., . . . Vizedom, A.

(2003). An interactive dialogue system for knowledge acquisition in cyc. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, (pp. 138–145).

Wu, T., He, S., Liu, J., Sun, S., Liu, K., Han, Q.-L., & Tang, Y. (2023). A brief overview of

ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10, 1122–1136.

Xu, W., & Forbus, K. (2025). Understanding Diagrams with Explicit Intermediate Visual

Representation. *Twelfth Annual Conference on Advances in Cognitive Systems*.

Yngve, V. H. (1961). Random generation of English sentences. *Proceedings of the International*

Conference on Machine Translation and Applied Language Analysis.

Zhao, K., & Forbus, K. (2025). Integrating Symbolic Natural Language Understanding and

Language Models for Word Sense Disambiguation. *arXiv preprint arXiv:2511.16577*.

Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J. Z., & Fredrikson, M. (2023). Universal and

transferable adversarial attacks on aligned language models. *arXiv preprint*

arXiv:2307.15043.