Compression and Abstraction Using Graph Based Meaning Representations

Anonymous ACL submission

Abstract

Graph-based meaning representations are widely used in NLP, where their abstraction level is determined once by dataset curators. Humans however, often use different levels of abstraction to adjust to different audience traits, like age or expertise. We develop methods to automatically adjust the abstraction level of graph based meaning representations to be more abstract or more granular. To get *more abstract* graphs, we develop an unsupervised pattern-finding and lossless graph-compression algorithm. We use this approach to compress the Process Execution Graph (PEG) dataset, and find semantically meaningful, cognitively-plausible patterns, leading to improved parsing precision (at the cost of recall). Finally, we present a case study for making representations of procedural texts *more granular*. We employ macro expansion to produce a challenging text-to-code dataset over the PEG graphs, decomposing predicates into their granular implementation. Taken together, we hope that this work will spur future research into better-suitable abstraction levels for different settings and scenarios.[1]

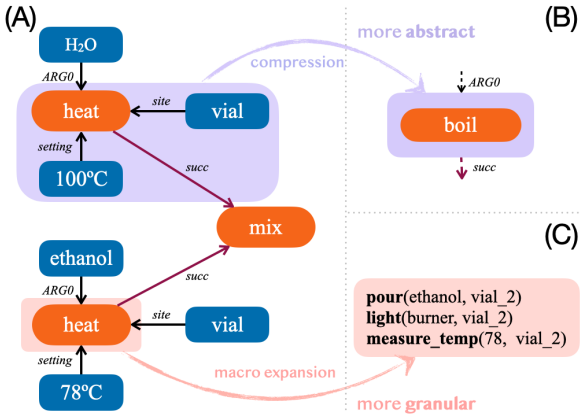


Figure 1: Exploring different levels of abstraction in a graph meaning representation of a wet lab protocol. (A) The full procedure at the level annotated by Tamari et al. (2021), containing two instances of the same subprocess of "boiling a reagent" (H_2O boiled in $100°C$, *ethanol* boiled in $78°C$), then mixing the outcomes; (B) a *more abstract* representation of the same process, achieved by compressing the subgraph colored in purple to a "boil" node in an unsupervised manner; (C) a *more granular* representation of the process, achieved by first converting the graph to code, then using *macro-expansion* to decompose the "heat" predicate to its granular implementation.

1 Introduction

Graph-based meaning representations are used in a wide range of NLP applications, such as broad coverage semantics or executable semantic representations (e.g. AMR, UD, or SQL; Banarescu et al., 2013; Nivre et al., 2016; Rubin and Berant, 2021). These meaning representations can be annotated in different levels of abstraction. For example, in Figure 1 the same wet lab process for *heating a liquid to its boiling temperature* can be described more *granularly*, e.g., specifying the temperature and the container for the liquid, or more *abstractly*, e.g., with the "boil" predicate. The choice of abstraction level for a given formalism is usually done by the task designers, and models then aim to reproduce that specific level of abstraction.

Exploring other abstraction levels is useful for two main reasons. First, from the cognitive perspective, humans often adapt their language according to their audience traits, e.g., their age or expertise. For example, different levels of abstractions will be used to describe a lab protocol to a layperson, a first-year undergrad student, or an expert biochemist (Dell and Brown, 2013). In regards to the wet lab process in Figure 1 for instance, we can describe it very specifically to a layerperson (part (C)), mention only the boiling temperature to an undergrand student (part (A)), and use only "boil" when communicating with an expert who knows the specific temperature (part (B)). Addi-

[1]We will make all resources publicly available.

tionally, creating abstractions by packing different operations in a single node can ensure that these operations occur in their entirety, e.g. turning off the fire can be an inherent part of boiling a liquid.

Second, more accurate meaning representation parsers may be obtained by training on a different abstraction level (Kate, 2008). For example, a higher abstraction level often results in simpler representations which reduces the search space of a model and allows it to predict smaller graph structures (i.e., containing less nodes and edges). On the other hand, using more granular representations can be beneficial as well, as it may help the model to identify semantic similarity between otherwise unrelated predicates.

The focus of this work is on abstracting representations through compression. To achieve this, we begin by developing an unsupervised lossless graph compression algorithm, based on the Minimum Description Length (MDL) principle (Rissanen, 1989). Our approach finds subgraphs which represent meaningful recurring patterns (e.g., boiling a liquid), and replaces instances of these patterns with a single pattern node to compress the gold training graphs (see Section 3). To test abstraction, we use the recently published Process Execution Graphs dataset (PEG; Tamari et al., 2021). PEG annotates wet lab protocols with executable meaning representation graphs, where each node represents either a predicate (e.g., "mix" or "heat"), or an argument (e.g., a "vial" or "water").

Our results and analysis (Section 4) show that semantically meaningful patterns can be found in an unsupervised fashion, and that their use for graph compression improves the precision of the PEG parser at the cost of recall.

Finally, we present a case study for making representations more granular (Section 5). To do this, we focus on executable meaning representations, such as Python programs or SQL. We perform *macro expansion*, i.e. replace each function call with its implementation, thus creating a longer, but more specific representation. We automatically convert PEG graphs into PEG2Code, a code format which can be run in a wet lab simulator, and serves as a challenging text-to-code dataset on its own.

To conclude, our main contributions are:

- We present a general unsupervised exploration of the abstraction level via unsupervised graph compression and macro expansion.
- We develop an unsupervised pattern-finding and lossless-compression algorithm applicable for various graph-based meaning representations, allowing us to find meaningful and effective patterns.
- We present a new text-to-code dataset (*PEG2Code*), containing complex examples of lab protocols in a natural language together with their corresponding execution code.

2 Background

In Section 2.1 we begin by introducing the Minimum Description Length (MDL) principle (Rissanen, 1989) for data representation and compression, and Subdue (Ketkar et al., 2005), a pattern-mining and compression algorithm which follows the MDL principle. Then in Section 2.2 we describe Process Execution Graphs (PEG; Tamari et al., 2021), on which we test our approach.

2.1 Unsupervised Graph Compression

MDL is a principle originating from information theory, which states that the best representation to describe a dataset is the one that minimizes the Description Length (DL) of the data, e.g. the number of bits to encode a graph. In this work we will adapt and extend Subdue (Ketkar et al., 2005), a MDL-based lossy graph-compression for directed acyclic unweighted graphs with labeled nodes and edges.

Subdue works as follows: starting from all the nodes with unique labels, at each iteration n it considers subgraphs of size $\leq n$ nodes, constructed from subgraphs that were considered in the $n-1$ iteration. It then ranks each subgraph according to its effectiveness in compressing the DL of the graphs, and saves the top K of them, to be extended and considered in the next iteration. The number of iterations and K are hyperparameters that we explore in this work. We name the chosen subgraphs for compression as *patterns*.

Each pattern consisting of nodes $U = \{u_1, \ldots, u_n\}$ can then be replaced with a single "pattern node" u'. Outgoing edges $(u_i, v); v \notin U$ in the original graph are replaced with an edge (u', v), and incoming edges (v, u_i) are similarly replaced with an edge (v, u'). Note that this is a *lossy compression* as it does not preserve information as the exact source or target nodes for either incoming or outgoing edges. E.g., for an incoming edge (v, u'), it is impossible to recover the original edge (v, u_i) with perfect certainty.

2.2 Process Execution Graphs (PEG)

PEG is a graph-based meaning representation, aiming to capture the predicate-argument structure of biochemistry lab protocols written in natural language. Formally, PEGs are labeled directed acyclic graphs, grounded in the original protocol. Each node corresponds to a tokens span in the text and represents either a predicate or an argument. Predicate is an action in the lab (for example, "heat"), while an argument is an object in the lab (e.g., a "vial"). In Figure 1 (A), predicate nodes are colored in orange, and argument nodes in blue. Tamari et al. (2021) annotated 279 protocols in English with corresponding PEG graphs, and have also presented a parser capable of predicting PEG from unstructured texts. Finally, PEG is coupled with a simulator which implements some of its predicates. These implementations will allow us to apply macro expansions, after we convert PEG graphs to code, to gain a more granular representation.

3 Unsupervised Graph Compression for Meaning Representations

We first describe our approach for getting *more abstract* representations using unsupervised graph compression. Following in Section 5 we present a case study for making a representation more granular.

Our compression-driven approach consists of the following steps (Figure 2): (1) given a parallel dataset of texts and corresponding graphs, we first find patterns in the graphs in an unsupervised manner, and use them to compress the graphs training set; (2) we then train a parsing model on the compressed graphs; (3) we use the trained model to predict compressed graphs for held-out texts; and finally (4) we decompress them back to the space of the original meaning representations. This allows us to compare the performance of our approach versus the original model. Figure 2a illustrates our training phase (steps 1-2), and Figure 2b illustrates our inference phase (steps 3-4), in comparison to original parsing paradigm.

In the following, we will first formalize our problem (Section 3.1), and then elaborate on our compression method (Section 3.2).

3.1 Formal Definitions

We formalize our problem as follows:

Input. Our input $G : \{g_i\}_{i=0}^{N}$ is a set of directed acyclic unweighted graphs with labeled nodes and edges. These requirements are general enough to hold for various prominent meaning representations, such as universal dependency, AMR, as well as executable formats such procedural code and SQL. In particular, in this work we will use the PEG graphs, which conform to these requirements and allow us to test both more granular and more abstract representations.

Output. We perform an unsupervised learning of frequent patterns H in G, and use them to compress all of its graphs $\{g_i\}_{i=0}^{N}$. A pattern $P \in H$ is a subgraph that appears in at least one of the graphs in G. Compression is done by replacing all the instances of every $P \in H$ in each graph $g \in G$ with one node per instance. We call this kind of node a *pattern node*. Formally, $\forall i \in [N]$, g_i' is the compressed graph of g_i, and our output is $G' := \{g_i'\}_{i=0}^{N}$ (see Figure 3 for example). Since we derive G' by only reducing subgraphs into nodes, it holds that the description length of G' is smaller or equal to that of the original set G.

3.2 Developing a Lossless Graph Compression

To prevent errors in decompression, we develop a lossless compression algorithm, by looking for patterns conforming with the following constraint: for a candidate pattern P, there are $v_{in}, v_{out} \in V(P)$ s.t. for every instance p of the pattern P in some graph g, it holds that:

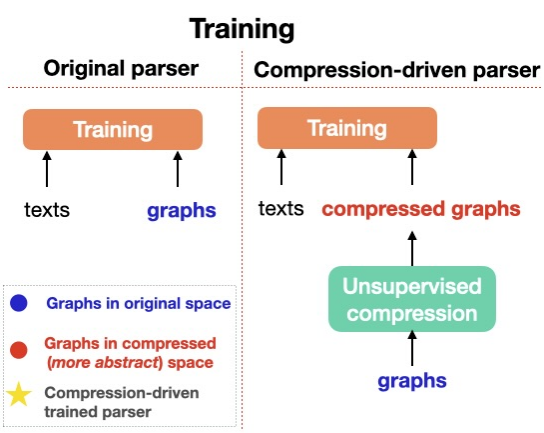
$$\begin{gathered}\forall (v_s, v_t) \in E(g), \\ (v_s \notin V(p) \wedge v_t \in V(p)) \Rightarrow v_t = v_{in}\end{gathered} \tag{1}$$

and

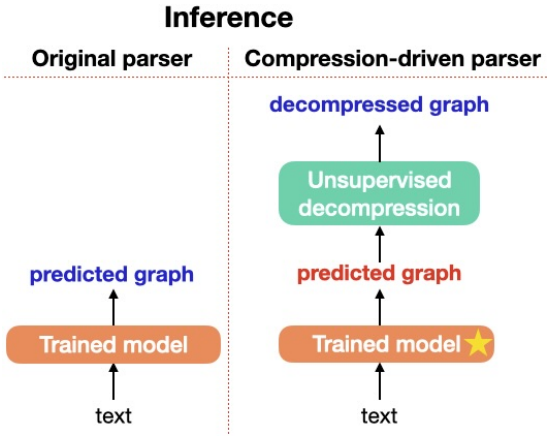
$$\begin{gathered}\forall (v_s, v_t) \in E(g), \\ (v_s \in V(p) \wedge v_t \notin V(p)) \Rightarrow v_s = v_{out}\end{gathered} \tag{2}$$

Intuitively, Equation 1 ensures that every incoming edge to the pattern is connected to the same node in the pattern, while Equation 2 ensures that every outgoing edge leaves the pattern from the same node.

When both conditions are met, an MDL based compression approach becomes lossless, as we can record a single *entry node* v_{in} and a single *exit node* v_{out} along with each pattern P. During decompression these two conditions ensure that we can connect every incoming edge to v_{in} and every outgoing edge to v_{out}. See Figure 3 for example.



(a) *Training phase*. Given a parser and a graph meaning representation dataset, instead of training on the original graphs (left side), we first apply compression, then train on graphs in the compressed, *more abstract* space (right side).



(b) *Inference phase*. Right side: the new (compression-driven) trained parser predicts a graph in the *more abstract* space, i.e. "compressed" graph. Then apply decompression to transfer the inferred graph to the original space, as the original model's inferred graphs (left side).

Figure 2: Experiment pipeline for our compression-driven parrser.

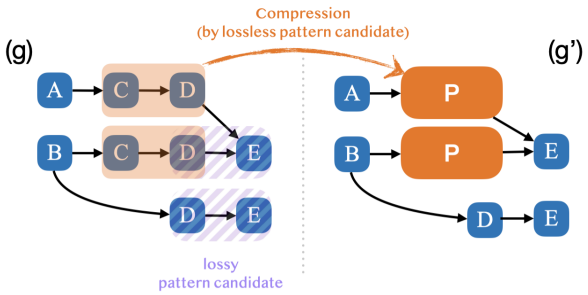


Figure 3: *Lossless graph compression example*. g (left) is a graph in the original space, containing two potential patterns: $(C \rightarrow D)$ (in orange) conforms to our lossless constraint in both of its instances, and is being compressed in g' (right) by replacing each instance with a *pattern node* P, where its $v_{in} = C$ and $v_{out} = D$. The other candidate $(D \rightarrow E)$ (in purple stripes) violates the constraint, as it has two incoming edges to **different** node labels: E (in the upper instance), and D (in the lower instance). Edge labels were omitted for simplicity.

Smoothed lossless constraint. The constraints in Equations 1 and 2 may sometimes be too strict. We therefore wish to relax them to allow for patterns that respect the constraint in general, apart from a few exceptional cases, e.g., due to noise or annotation error. We implemented this relaxation by setting a threshold T, which represents the maximum allowed violations of the constraint for each pattern.[2]

[2]Out of computational considerations we checked the number of instances that violated the constraints, regardless the number of violations in each instance.

4 Evaluation

In this section we first describe our evaluation and experimental setup for applying our compression method on the PEG dataset. Following we show our main results for parsing PEG (Section 4.1) and analysis insights in regards to our compression paradigm (Section 4.2).

Evaluation setup. We evaluate our compression approach on PEG graphs, as outlined in Figure 2. Most importantly, since our compression is lossless, we convert the predicted compressed graphs back to the space of the original PEG graphs (Figure 2b), and thus are able to compare a model trained on the original graphs versus a model trained on the compressed graphs. We calculate precision, recall and F1 on the edges of the decompressed predicted graphs, considering the edge label and both its source and target nodes' labels. In a preliminary experiment we find that the variability between different training seeds is small: ± 0.09 on average over all edge labels.

Experimental Setup The PEG dataset consists of 279 annotated protocols. We follow Tamari et al. (2021), and use them in a 5-fold cross validation: 2 folds (112 protocols) for training, and 3 folds (167 protocols) for evaluation. We use each set of chosen patterns to compress the data and train the same model architecture on 4 RTX 2080Ti GPUs (8 hours of training on average). We follow the same configuration and hyperparameters that were used by Tamari et al. (2021) for training.

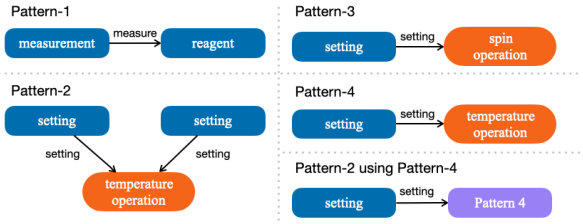


Figure 4: *Patterns found on PEG graphs*. Pattern-1: a reagent in a specific amount; Pattern-2: a temperature operation (e.g. "heat") with two of its settings (e.g. $87°C$, "overnight"); Pattern-3: spin operation with its setting; Pattern-4: temperature operation with one setting; Pattern-2 using Pattern-4: A demonstration of recursive patterns, where Pattern-2 includes Pattern-4.

4.1 Compression and Parsing Results

Our parsing results are presented in Table 1. The results in this table are for graphs in the original PEG graphs space (i.e. for evaluation after decompressing the inferred graphs), which are comparable to the results presented by Tamari et al. (2021); and Table 3 presents results for an additional evaluation in the compressed graphs space. The patterns that were used for compression are illustrated in Figure 4. Below we outline key observations drawn from these results.

Training the parser on compressed graphs improves precision ... The patterns that were found by our lossless compression algorithm and were used to compress PEG graphs improve the overall parsing precision. This improvement is notably seen for the most common "ARG0" edge label, that was improved by +2.41% after using Pattern-3 for compression, as well as for the other common important "measure" edge label, which was improved by +8.46% after using Pattern-1. For all the edge labels except "ARG2" and "co-ref-of", the best precision per edge label was achieved by one of the models that were trained on compressed graphs (see bold-highlighted numbers in Table 1).

...but decreases recall The use of compressed graphs for training decreases the model's recall. As pattern nodes are considerably less frequent in comparison to regular nodes in the graphs, we assume that the model is less likely to predict them. As a result, the edge labels that are contained in these pattern nodes are predicted less than the original graph distribution. A possible strategy to improve the recall is thus using more frequent patterns for compression by omitting the lossless constraint. This would require a strategy for decompression, for example, identifying the most likely source and target node for each edge type. Alternatively, we can consider modifying the MDL objective to prefer more frequent patterns. Finally, future work may consider representing both pattern nodes as well as their original subgraphs within the same graph representation.

Improved precision and decreasing recall are notably seen for edges that touch pattern nodes. We analyze two types of edges in the compressed predicted graphs: (1) *pattern edge*: an edge (u, v) where either u or v is a compressed pattern node; and (2) *normal edge*: an edge that none of its nodes are pattern nodes. The results on the compressed graphs (Table 3) show that the use of patterns from our lossless compression algorithm gives a high precision for pattern edges, and increasingly so when using more patterns. The recall of the same edges is low, and generally gets lower when using more such patterns. This can be one prominent cause of the general observation of low recall.

These results can be interpreted as baseline results for parsing into the compressed graphs space. We observe that specially for pattern edges of type "ARG0" the precision is notably high (79.12-81.06). Additionally, the precision of "ARG2" pattern-edges is between 83.56-100.00, except when compressing by Pattern-1 only.

4.2 Analysis

Semantically meaningful patterns were found in an unsupervised manner. A manual examination of the compressed patterns reveals meaningful patterns. For example, see the patterns illustrated in Figure 4. Even though the patterns were found in an unsupervised manner, they reveal domain-specific knowledge, e.g. that a reagent should be mentioned with its amount (measure), and that a temperature operation like "heat" requires at least one or two settings, like *what* to heat and *to what temperature*.

Recursive patterns were found in the data. Our lossless compression algorithm finds recursive patterns, i.e. patterns that contain previously-found patterns. For example see Pattern-2, which contains Pattern-4 and can be represented by using it, as shown in Figure 4 in "(Pattern-2 using Pattern-4)". Recursive patterns are an interesting phenomenon, as it can indicate a modular structure of the graphs, and known to be a unique and fundamental feature

Edge label	Original model			Pattern 1			Pattern 2			Pattern 3			Pattern 4		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
All Labels	61.12	65.92	63.43	61.88	62.02	61.95	**62.07**	61.32	61.70	61.64	62.80	62.21	61.22	62.98	62.09
ARG0	64.95	70.10	67.43	64.92	66.11	65.51	66.35	67.16	66.75	**67.36**	68.57	67.96	65.42	67.16	66.28
ARG1	49.84	46.50	48.11	46.79	43.59	45.13	**51.40**	48.10	49.70	48.08	40.23	43.81	50.00	45.34	47.55
ARG2	**81.82**	69.57	75.20	81.14	68.60	74.35	81.77	71.50	76.29	79.55	33.82	47.46	78.57	69.08	73.52
co-ref-of	**65.98**	62.49	64.19	65.51	55.71	60.22	65.63	66.02	65.83	65.95	66.73	66.33	64.81	61.84	63.29
located-at	30.07	17.04	21.75	34.69	6.30	10.66	**35.15**	21.48	26.67	34.98	26.30	30.02	28.77	15.56	20.19
measure	71.77	86.49	78.44	**80.23**	73.70	76.83	73.29	87.85	79.91	71.28	85.70	77.83	74.20	84.49	79.01
modifier	51.64	62.08	56.38	**53.90**	57.89	55.82	53.87	59.93	56.74	53.04	59.57	56.11	52.20	59.57	55.64
part-of	33.93	14.07	19.90	25.00	12.59	16.75	**35.94**	17.04	23.12	32.84	16.30	21.78	29.73	16.30	21.05
setting	71.87	78.35	74.97	**71.90**	78.58	75.09	70.27	55.47	62.00	70.95	65.59	68.17	70.25	71.45	70.84
site	61.40	66.07	63.65	57.86	62.09	59.90	61.41	67.36	64.25	62.39	68.28	65.20	**63.28**	65.34	64.29
succ	54.71	63.13	58.62	56.55	62.62	59.43	**56.72**	53.32	54.97	54.89	57.19	56.02	53.74	58.66	56.10
usage	33.80	38.01	35.78	33.82	31.77	32.76	32.85	35.48	34.11	34.01	35.87	34.91	**35.85**	36.06	35.96

Table 1: *Parsing results on decompressed graphs*, predicted by models that were trained on compressed PEG graphs. We show the Precision (P), Recall (R), and F1 - per edge label and across labels ("All Labels"). "Original model" shows the results of the model that trained on the original (non-compressed) graphs. 'Pattern i' column ($i \in [1, 4]$) presents the results of a model that was trained on graphs which are compressed by Pattern-i. The best precision per edge label is highlighted in bold.

of human cognition (Hauser et al., 2002; Corballis, 2007). This property may allow an adjustable increase in the abstraction level, by choosing which depth of the patterns to use.

There are few violations of the lossless constraint. As mentioned in Section 3.2, we apply a smoothing to our lossless constraint, so potential patterns can violate it up to some predefined threshold. We run our lossless compression algorithm with different smoothing thresholds (from 5 to 30), finding that its chosen patterns respect the constraint in 99.96% of the times.[3] This indicates that our constraint for procedural processes is inherently represented in the PEG dataset. Future work may explore if this holds for other procedural texts as well.

5 Case Study: Making Executable Representations more Granular

So far we presented an approach for deriving more abstract representations via graph compression. Complementing this exploration is making representations more granular. This can be beneficial for presenting information to laypersons, or for parsing accuracy, as a more granular representation can surface semantic similarities between predicates which share similar granular components. We present a case study to demonstrate this direction on executable texts.

In Section 5.1 we outline a method to obtain more granularity for executable representations via macro expansion, and in Section 5.2 we implement this approach for the PEG dataset, producing PEG2Code, a challenging text-to-code dataset in various abstraction levels.

[3]Only 5 instances were found to violate the constraint.

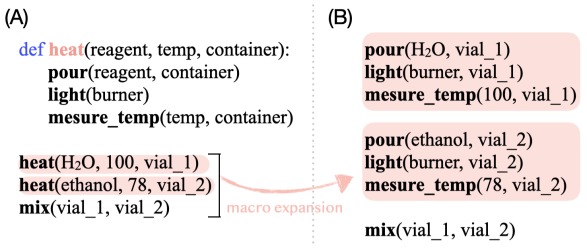


Figure 5: *Macro expansion example.* (A) Original code with *heat* function definition and its function-calls. (B) Code after applying macro expansion. We expand *heat* function's definition, and replace its input argument names with the variable names from the function call.

5.1 Granularity via Macro Expansion

An important subset of graph based meaning representations is that where the resulting graphs can be *executed*, often referred to as semantic parsing. Prominent executable representations include text-to-SQL (e.g. Yu et al., 2018), where the produced outputs can be queried against a database, or more generally, text-to-code (e.g. Desai et al., 2016), where the output can be run by python interpreter, for example.[4]

[4]Note that in both text-to-SQL and text-to-code, the output is equivalent to an abstract syntax tree (AST) which is a specific form of directed graph.

Abstraction in executable representations is often obtained via encapsulation of actions within subprocedures, such as functions or macros. A call to a subprocedure can then replace the more granular sequence of constituent actions, creating a more abstract representation. For example in Figure 5, the call to "heat" function in (A) can replace the sequence of actions colored in (B).

Consequently, given an executable representation containing such subprocedure, we suggest instantiating them with their implementation (i.e., macro expansion) to obtain a more granular, but semantically-equivalent, representation. E.g. the transition from (A) to (B) in Figure 5. Finally, it is possible to "adjust" the abstraction level by choosing which subprocedure to expand.

5.2 Creating the PEG2Code Dataset and Applying Macro expansion

We present a new dataset, called PEG2Code, to demonstrate the use of macro expansion for attaining more granular executable representation. PEG2Code creation process is exemplified in this section using Figure 6, where its edges are mentioned by the number of their traversal order, colored in yellow.

Creating PEG2Code. PEG2Code views PEG protocols as abstract syntax trees for procedural texts. PEGs have three types of edges: pred-arg, pred-pred and arg-arg edges. To convert a graph into code, its edges are traversed in a topological order. Below we outline the text-to-code interpretation for each of the edge types:

(1) *pred-arg* edges are interpreted as an input argument to a function, where the edge label specifies the type of the argument. For example, edge number 2 in the figure represents an "ethanol" input argument of type "ARG0" to a "heat" predicate, and affects the third code line (right). (2) *pred-pred* edges are interpreted as a function composition, and also define a temporal ordering between functions. For example, edge 4 defines that the "heat" predicate precedes the "mix" predicate and serves as its input. (3) *arg-arg* edges are interpreted as a "setter" in object oriented programming, i.e. updating a field of an object, where the edge label specifies the name of that field. For example, edge 1 sets the "measure" field of the "ethanol" object to 9, results in the second code line.

When we encounter an argument node for the first time, an instantiation command is generated

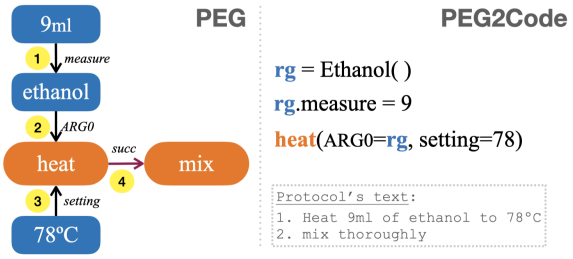


Figure 6: PEG graph (left) to code (right) conversion example, as used to create PEG2Code. The text of the protocol is presented in the lower right corner. The topological order of edges is presented in yellow circles. Predicates are colored in orange, arguments are colored in blue. Mix predicate conversion was omitted for simplicity.

to create its object in the PEG2Code program. For instance, visiting edge 1 generates an "ethanol" object instansiation, which is the first code line. Visiting a pred-pred edge (u, v) results in a function-call command to the predicate represented by the source node u. This function-call contains all the input arguments that were "collected" from all the incoming pred-arg edges, (w, u) to that predicate node u. For example, edge 4 triggers the third code line, i.e. calling the "heat" function with "ethanol" as the "ARG0" input argument and 78 as the "setting" input argument (as determined by edges 2,3 that enter into the "heat" node).

Applying macro expansion in PEG2Code. Conveniently, each PEG's predicate defines a sequence of primitive predicates, which keep track of objects in a simulated lab environment. This allows us to present PEG2Code in a more granular format by performing macro expansions where the primitive code is available (see Figure 5 for an example).

Comparing PEG2Code to other text-to-code datasets. Table 2 compares PEG2Code to other popular text-to-code datasets, namely CoNaLa (Yin et al., 2018), and CONCODE (Iyer et al., 2018). As shown in the table, other datasets consist of short and simple code snippets, e.g. 7.37 tokens in CoNaLa, and 26.3 tokens in CONCODE on average. In addition, their text (i.e. code documentation) is more templated (e.g. "compare contents at filehandles 'file1' and 'file2' using difflib").

In contrast, the text in PEG2Code is in natural language (see example in Figure 6), and the code in each example is considerably longer (138.94 tokens on average), however it consists of small vocabulary and command types. The command

Dataset name	# Examples	Avg. NL tokens	Avg. code characters	Avg. code tokens
CoNaLa (Python only)	37,719	13.20	65.40	7.37
CONCODE (Java)	104,000	13.73	119.00	26.30
PEG2Code (No IDs*)	279	194.03	744.16	138.94

Table 2: Quantitative comparison between Text-to-Code datasets. PEG2Code consists of longer Natural Language (NL) texts, measured by the average number of tokens per example, and code, measured by the average number of code tokens (and characters) per example. * We use a version of PEG2Code without unique IDs in variable names, so the measured length of the code would not be affected by these IDs.

types in PEG2Code are function calls, instantiations, assignments and setters. It lacks control-flow statements, as it originates from a composition of functions, with no conditions or loops.

Fine-tuning T5 for text-to-code with PEG2Code dataset does achieves only negligible BLEU results We fine-tuned T5 model (Raffel et al., 2019), to test it as a baseline text-to-code model on PEG2Code data, as well as on the macro-expansioned version of PEG2Code. An evaluation of these models gives low BLEU scores. Recent models show low performance on other popular text-to-code datasets as well, e.g. CoNaLa seq2seq baselines performance ranges between 10.58-14.26 BLUE scores. As the PEG2Code dataset is more challenging than existing datasets, we expected the given low results from fine-tuned T5. Future work may explore dedicated architectures to tackle this hard problem.

6 Related Work

As far as we know, we are the first to try the suggested exploration of representing data in different abstraction levels, combining pattern mining, graph compression (or macro expansion) and language.

Most related are works that deal with graph compression in general, sometimes termed graph coarsening (Cai et al., 2021). Sutton et al. (1999) represent knowledge at changing levels of temporal abstractions; El-Kilany et al. (2012) develop sentence compression by UD graph pruning; Filippova and Strube (2008) introduce sentence fusion by dependency graph compression, and Kate (2008) transforms grammars to match NL semantics. Other works that deal with pattern finding are usually either based in graph theory (Inokuchi et al., 2000), or data mining (Agrawal and Srikant, 1995).

Other works find patterns in natural text (Califf and Mooney, 2003; Shnarch et al., 2017), but differently from us they use either initial syntactic templates, or supervised learning. Goldman et al. (2018) converted programs and utterances that describe them into a predefined abstract form, and a recent work by Wong et al. (2021) use language to learn program abstractions and search heuristics.

Additional recent and similar works can be found in the fields of vision and graph neural networks. For example, Edwards et al. (2020) use multi-resolution sampling and graph convolutional neural network for multi-scale feature learning. Another examples are the use of multiscale graph neural networks (Li et al., 2020; Xu et al., 2019). Differently from us, these works mainly deal with undirected and unlabeled graphs.

7 Conclusion

We introduced an approach for unsupervised exploration of different abstraction levels of graph-based meaning representations. We focused on making graph representations *more abstract*, and developed a pattern-finding and lossless-compression algorithm, suitable for general labeled directed acyclic graphs. We created an abstracted representation of PEG graphs, and showed that semantically meaningful patterns can be found in an unsupervised fashion, and that by using them for compressing PEG graphs we can improve its parser precision, at the cost of recall. Finally, we presented a case study to achieve *more granular* representations. We created a text-to-code dataset from PEG, by applying *macro expansion* on its predicates, resulting in more primitive predicates.

We hope that this work will inspire future research into abstraction levels for a variety of situations and settings, and that the methods and dataset will serve as effective resources.

Limitations

We currently find only continuous linear patterns. Our lossless compression algorithm works in a bottom-up manner, as it adds each time an

edge (and its other node) to an existing pattern-candidate. As a result, we end up with continuous linear patterns, i.e. full subgraphs. That means we currently can't find pattern that starts and ends with specific node labels, and contains a *varied* node label between them. We assume that covering these kind of patterns can lead to interesting structures, which can improve the performance. In order to do so, a new finding-patterns algorithm should be added, and we leave that for a future work (by us or others).

Approximated grounding for pattern nodes. PEG's model learns jointly both the structure of the graphs and their grounding to the given text. In the annotated data there is a text span as a grounding for each node. For pattern node, we chose to take the span between the left most and right most tokens among the spans of the nodes consisting it. Comparing to the annotated grounding, this approximation is less accurate. It might harm the performance of the models that were trained on the compressed data (so we challenged our models comparing the original model, that was trained with more accurate grounding).

Our method for achieving granular representations is applicable only for executable graphs. While converting the graphs into code, we leveraged the properties of executable graph (specifically demonstrated on PEG's graphs), e.g. that each node can be interpreted as either a predicate, or a predicate argument. Hence, this method for implementing the approach of achieving more granular representation of graph meaning representation is limited to this kind of graphs.

PEG dataset is domain-specific. We chose to demonstrate our approach of exploring different abstraction levels of data on PEG, as its rich annotation allowed us to check both *more abstract* and *more granular* methods. Having saying that, PEG is limited to the domain of biochemistry. Its vocabulary size is 6394 words, and it contains 279 examples. Our approach should be applicable beyond these dataset and domain, but we haven't tested it yet.

References

R. Agrawal and R. Srikant. 1995. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*, pages 178–186.

Chen Cai, Dingkang Wang, and Yusu Wang. 2021. Graph coarsening with neural networks. *arXiv preprint arXiv:2102.01350*.

Mary Elaine Califf and Raymond J Mooney. 2003. Bottom-up relational learning of pattern matching rules for information extraction. *Journal of Machine Learning Research*, 4(Jun):177–210.

Michael C Corballis. 2007. The uniqueness of human recursive thinking: the ability to think about thinking may be the critical attribute that distinguishes us from all other species. *American Scientist*, 95(3):240–248.

Gary S Dell and Paula M Brown. 2013. Mechanisms for listener-adaptation in language production: Limiting the role of the" model of the listener". In *Bridges Between Psychology and Linguistics*, pages 117–142. Psychology Press.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356.

Michael Edwards, Xianghua Xie, Robert I Palmer, Gary KL Tam, Rob Alcock, and Carl Roobottom. 2020. Graph convolutional neural network for multi-scale feature learning. *Computer Vision and Image Understanding*, 194:102881.

Ayman El-Kilany, Samhaa El-Beltagy, and Mohamed E. El-Sharkawi. 2012. Sentence compression via clustering of dependency graph nodes.

Katja Filippova and Michael Strube. 2008. Sentence fusion via dependency graph compression. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 177–185.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. 2018. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1809–1819, Melbourne, Australia. Association for Computational Linguistics.

Marc D Hauser, Noam Chomsky, and W Tecumseh Fitch. 2002. The faculty of language: what is it, who has it, and how did it evolve? *science*, 298(5598):1569–1579.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *European conference on principles of data mining and knowledge discovery*, pages 13–23. Springer.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Rohit Kate. 2008. Transforming meaning representation grammars to improve semantic parsing. In *CoNLL 2008: Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 33–40, Manchester, England. Coling 2008 Organizing Committee.

Nikhil S Ketkar, Lawrence B Holder, and Diane J Cook. 2005. Subdue: Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 71–76.

Maosen Li, Siheng Chen, Yangheng Zhao, Ya Zhang, Yanfeng Wang, and Qi Tian. 2020. Dynamic multiscale graph neural networks for 3d skeleton based human motion prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 214–223.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, Portorož, Slovenia. European Language Resources Association (ELRA).

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

Jorma Rissanen. 1989. Stochastic complexity in statistical inquiry. *World scientific series in computer science*, 15:79–93.

Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive bottom-up semantic parsing. In *Proceedings of the 5th Workshop on Structured Prediction for NLP (SPNLP 2021)*, pages 12–21, Online. Association for Computational Linguistics.

Eyal Shnarch, Ran Levy, Vikas Raykar, and Noam Slonim. 2017. GRASP: Rich patterns for argumentation mining. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1345–1350, Copenhagen, Denmark. Association for Computational Linguistics.

Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211.

Ronen Tamari, Fan Bai, Alan Ritter, and Gabriel Stanovsky. 2021. Process-level representation of scientific protocols with interactive annotation. *arXiv preprint arXiv:2101.10244*.

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. 2021. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pages 11193–11204. PMLR.

Mingxing Xu, Wenrui Dai, Yangmei Shen, and Hongkai Xiong. 2019. Msgcnn: Multi-scale graph convolutional neural network for point cloud segmentation. In *2019 IEEE Fifth International Conference on Multimedia Big Data (BigMM)*, pages 118–127. IEEE.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pages 476–486. IEEE.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

A Appendix

Edge type	Edge-label	Pattern 1			Patterns 1-2			Patterns 1-3			Patterns 1-4		
		P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
PE	**All labels**	64.99	26.87	38.02	65.06	20.23	30.86	71.36	25.57	37.65	70.49	23.70	35.47
	ARG0	79.34	39.93	53.13	81.06	31.04	44.89	81.05	31.92	45.81	79.12	29.13	42.58
	ARG1	28.81	15.04	19.77	28.81	14.53	19.32	38.10	16.16	22.70	48.57	16.67	24.82
	ARG2	50.00	4.00	7.41	100.00	4.00	7.69	83.56	35.26	49.59	84.29	33.71	48.16
	co-ref-of	56.70	15.15	23.91	55.37	18.46	27.69	56.57	15.43	24.24	54.17	17.91	26.92
	located-at	0.00	0.00	0.00	10.00	2.67	4.21	13.04	4.00	6.12	6.25	1.33	2.20
	measure	35.71	6.49	10.99	33.33	5.19	8.99	28.00	4.55	7.82	56.25	5.84	10.59
	modifier	54.17	12.50	20.31	50.00	11.93	19.26	51.72	12.82	20.55	50.00	11.02	18.06
	part-of	47.06	26.67	34.04	72.73	26.67	39.02	88.89	26.67	41.03	72.73	26.67	39.02
	setting	N/A	N/A	N/A	0.00	0.00	0.00	94.53	48.02	63.68	96.03	47.27	63.35
	site	53.91	33.51	41.33	54.92	34.54	42.41	56.64	30.62	39.75	55.64	26.33	35.75
	succ	N/A	N/A	N/A	0.00	0.00	0.00	72.73	21.60	33.31	66.48	18.85	29.38
	usage	66.67	17.39	27.59	80.00	11.43	20.00	33.33	7.55	12.31	54.55	9.38	16.00
NE	**All labels**	58.34	62.53	60.36	55.97	62.71	59.15	53.96	59.56	56.62	52.95	58.72	55.69
	ARG0	58.52	64.14	61.20	59.72	64.48	62.01	57.91	63.76	60.69	56.62	62.26	59.31
	ARG1	51.32	43.98	47.37	47.64	49.56	48.58	41.93	43.65	42.77	44.06	39.21	41.49
	ARG2	80.47	74.73	77.49	80.23	75.82	77.97	54.55	17.65	26.67	36.36	12.50	18.60
	co-ref-of	61.63	65.32	63.42	56.24	71.61	63.00	61.45	68.54	64.80	57.98	69.14	63.07
	located-at	30.34	13.85	19.01	34.78	20.51	25.81	32.31	21.54	25.85	35.19	19.49	25.08
	measure	60.26	65.28	62.67	62.77	67.13	64.88	62.45	70.83	66.38	60.17	67.13	63.46
	modifier	52.03	61.20	56.25	52.02	60.25	55.83	50.06	62.17	55.46	51.09	62.62	56.27
	part-of	17.91	11.43	13.95	28.81	16.19	20.73	24.62	15.24	18.82	30.14	20.95	24.72
	setting	72.63	77.47	74.97	64.96	74.00	69.18	52.43	60.18	56.04	45.19	51.87	48.30
	site	61.83	66.02	63.86	60.16	68.66	64.13	59.96	67.14	63.35	58.20	68.94	63.12
	succ	55.22	62.29	58.54	53.20	60.58	56.65	51.51	57.37	54.28	51.46	56.74	53.97
	usage	31.84	33.27	32.53	29.95	35.15	32.34	32.22	35.65	33.85	33.47	37.64	35.43

Table 3: *Evaluation on compressed inferred graphs* (before decompression), separated by edge type ('PE': pattern-edge, 'NE': normal edge), and by edge label. We show Precision (P), Recall (R) and F1 for the predictions of the models that was trained on compressed graphs. 'Pattern(s) $1(-x)$' column: the results for the model that was trained on compressed graphs by pattern(s): 'Pattern-1', 'Pattern2', ..., 'Pattern-x'. (N/A is assigned where there are no edges of that type to predict in the gold graphs, and no False Positives)