Sequence-to-sequence AMR Parsing with Ancestor Information

Chen Yu and **Daniel Gildea**
Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

AMR parsing is the task of mapping a sentence to an AMR semantic graph automatically. The difficulty comes from generating the complex graph structure. The previous state-of-the-art method translates the AMR graph into a sequence, then directly fine-tunes a pretrained sequence-to-sequence Transformer model (BART). However, purely treating the graph as a sequence does not take advantage of structural information about the graph. In this paper, we design several strategies to add the important *ancestor information* into the Transformer Decoder. Our experiments[1] show that we can improve the performance for both the AMR 2.0 and AMR 3.0 dataset and achieve new state-of-the-art results.

1 Introduction

Abstract Meaning Representation (AMR) (Banarescu et al., 2013) is a graph that encodes the semantic meaning of a sentence. In Figure 1a, we show the AMR of the sentence: *You told me to wash the dog*. AMR has been widely used in many NLP tasks (Liu et al., 2015; Hardy and Vlachos, 2018; Mitra and Baral, 2016).

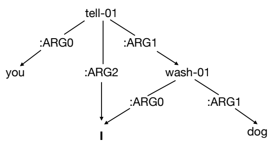
AMR parsing is the task of mapping a sentence to an AMR semantic graph automatically. A graph is a complex data structure which is composed of multiple vertices and edges. There are roughly four types of parsing strategies in previous work:

- **Two-Stage Parsing** (Flanigan et al., 2014; Lyu and Titov, 2018; Zhang et al., 2019a; Zhou et al., 2020): first produce vertices, and produce edges after that.

- **Transition-Based Parsing** (Damonte et al., 2016; Ballesteros and Al-Onaizan, 2017; Guo and Lu, 2018; Wang and Xue, 2017; Naseem et al., 2019; Astudillo et al., 2020; Zhou et al., 2021): process the sentence from left to right, and produce vertices and edges based on the current focused word.

- **Graph-Based Parsing** (Zhang et al., 2019b; Cai and Lam, 2019, 2020): produce vertices and edges based on a graph traversal order, such as DFS or BFS.

- **Sequence-to-Sequence Parsing** (Konstas et al., 2017; van Noord and Bos, 2017; Peng et al., 2017, 2018; Xu et al., 2020; Bevilacqua et al., 2021): this method linearizes the AMR graph to a sequence, then uses a sequence-to-sequence model to do the parsing.



(a)

(<R0> tell-01 :ARG0 (<R1> you)
:ARG2 (<R2> I)):ARG1 (<R3> wash-01
:ARG0 <R2> :ARG1 (<R4> dog)) }}

(b)

Figure 1: AMR Graph and linearization for the Sentence: *You told me to wash the dog.*

Bevilacqua et al. (2021) achieved the state-of-the-art performance by using the last seq-to-seq strategy. They linearized the AMR graph (see Figure 1b) and fine-tuned BART (Lewis et al., 2020), a denoising sequence-to-sequence pretrained model based on Transformer (Vaswani et al., 2017), for the parsing. We briefly show the method in Figure 2. During training, they linearize all the AMR graphs in the training dataset into sequences, then they can fine-tune the BART model in this new sequence-to-sequence dataset. At inference time, they first generate the AMR sequence using the BART model, then they recover the AMR graph from this sequence.

However, purely treating the graph as a sequence may not take advantage of important information about the structure of the graph. When generating

[1] `https://github.com/lukecyu/amr-parser-s2s-ancestor`

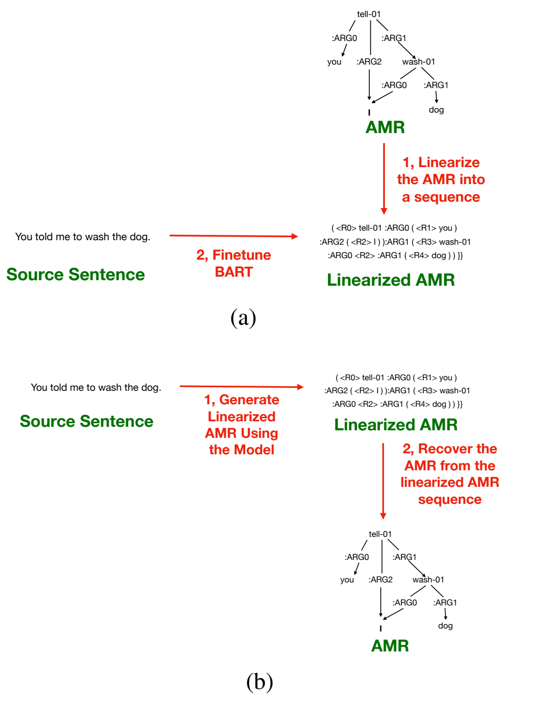

tell-01
:ARG0
:ARG1
you
:ARG2
wash-01
:ARG0
:ARG1
I
dog
AMR
1, Linearize the AMR into a sequence
You told me to wash the dog.
2, Finetune BART
Source Sentence
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)):ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog)) }}
Linearized AMR
(a)
You told me to wash the dog.
Source Sentence
1, Generate Linearized AMR Using the Model
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)):ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog)) }}
Linearized AMR
2, Recover the AMR from the linearized AMR sequence
tell-01
:ARG0
:ARG1
you
:ARG2
wash-01
:ARG0
:ARG1
I
dog
AMR
(b)

Figure 2: AMR Graph and linearization for the Sentence: *You told me to wash the dog.*

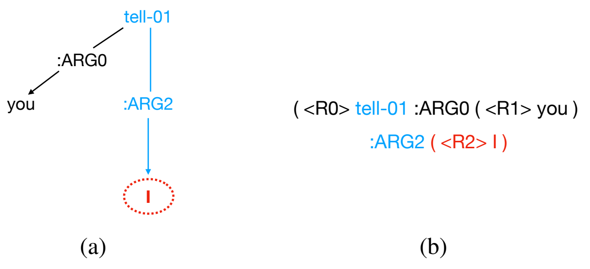


Figure 3: Example of finding Ancestors.

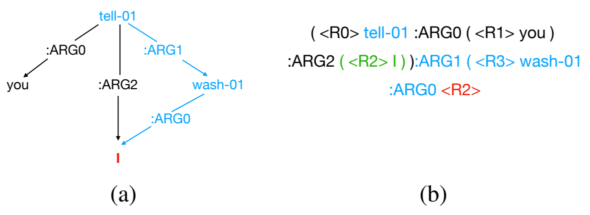

tell-01
:ARG0
:ARG1
you
:ARG2
wash-01
:ARG0
I
(<R0> tell-01 :ARG0 (<R1> you)
:ARG2 (<R2> I)):ARG1 (<R3> wash-01
:ARG0 <R2>
(a)
(b)

Figure 4: Example of finding ancestors with re-entrancy.

the last token *dog* in Figure 1b, for example, the dot-product attention layer in the Transformer Decoder attends to all the previous tokens and lets the model learn the weight of these tokens. However, if we can tell the model which tokens are its ancestors, like its parent is *wash-01* and its grand-parent is *tell-01* (see Figure 1a), it will make this token much easier to generate. Adding graph structure has been demonstrated to be useful for the AMR-to-text task (Zhu et al., 2019; Yao et al., 2020; Wang et al., 2020). These approaches added the graph structure to the Transformer Encoder. Therefore, we expect that adding structure in **Transformer Decoder** for AMR parsing task will also be helpful.

In this paper, we base our work on the seq-to-seq model of Bevilacqua et al. (2021) with the AMR linearized by DFS traversal order. We introduce several strategies to add ancestor information into the Transformer Decoder layer. We also propose a novel strategy, which consists of setting parameters in the mask matrix for those ancestor tokens and tuning them. We find that this new strategy makes the largest improvement.

2 Add Ancestors Information into Model

2.1 DFS linearization and Ancestors

The DFS linearization of Bevilacqua et al. (2021) used pairs of parentheses to indicate the start and the end of exploring a node in the DFS traversal order. The readers can use Figure 1 as an example and are referred to Bevilacqua et al. (2021) for more details.

This means when generating the next token, we can construct the partial graph from previous tokens and determine the ancestors tokens among them. In Figure 3b, for example, when we generate the token *I*, we can construct the partial graph in Figure 3a and find its ancestors (*tell-01 –> :ARG2 –>*).

If AMR were a tree, then the ancestors of each token would be clear to define. However, since AMR is a graph, one node may be visited multiple times (which is called re-entrancy), which brings ambiguity to find the ancestors. For example, in Figure 4, when we generate the last token *<R2>*, it is actually the re-entrancy of the token *I* generated before. Under this circumstance, we will use the tokens in the new path (*tell-01 –> :ARG1 –> wash-01 –> :ARG0 –>*) as its ancestors. We cannot use tokens from the old path (*tell-01 –> :ARG2 –>*), since we cannot know it is a re-entrancy before we have actually generated it.

2.2 Transformer Background

The original Transformer (Vaswani et al., 2017) used scaled dot-product self-attention. Typically, the input of the attention consists of a query matrix Q, a key matrix K and a value matrix V, the columns of which represent the query vector, the key vector and the value vector of each token. The attention matrix can be calculated as follows:

$$\text{Attention}(Q, K, V, M) = \text{Softmax}\left(\frac{S}{\sqrt{d}} + M\right) V,$$
$$S = QK^{\top},$$

where $Q, K, V \in \mathbb{R}^{N \times d}$, N is the length of the sequence, d is the dimension of the model, and M is the **mask matrix** to control which tokens in the sequence are attended for a given token.

A typical Transformer module consists of several layers. In each layer it uses MultiHead attention. For each head, it calculates attention as above, and then averages the results.

In the Encoder self-attention and Encoder-Decoder attention layers, the mask matrix is the same across all the heads and all the layers, and all the elements in the matrix are 0, meaning all the tokens are attended. But in the Decoder self-attention layers, the elements denoting the attention to the future token ($M_{i,j}$ with $i < j$) are set to $-\infty$, meaning that they have no effect when calculating the weighted sum.

2.3 Add Ancestor Information into Model

We focus on the **mask matrix** M in the Transformer Decoder self-attention layers to add the ancestor information during the parsing. We introduce two strategies: a hard strategy and a novel soft strategy.

Hard Strategy Under this strategy, we set elements denoting the ancestors to 0, and the elements denoting the non-ancestors to $-\infty$ in M, such that only the ancestor tokens are attended. We will explore the influence by using the new mask matrix only on some decoder layers or on some heads.

Soft Strategy Under this novel strategy, we will not mask the non-ancestor tokens and abandon them in a hard way. Instead, what we do is only telling the model which are the ancestor tokens and letting the model learn the weights by itself. Specifically, we use three different values in the mask matrix: $-\infty$ for all future tokens; 0 for all non-ancestor previous tokens; parameter α for all ancestor tokens. We let the model learn the weight α to control how much it should focus on the ancestor tokens. Similar to the hard strategy, we will also explore the influence by setting different parameters on different layers or on different heads.

2.4 Inference

During the inference stage, the input of the decoder is no longer the complete linearized AMR sequence. Instead, it is dynamically extended, and, at each step, the input is the tokens that have been generated during the previous steps. A natural question is: how can we find the ancestors of a token when we don't yet have a complete sequence (and therefore can't convert it to a graph to find its ancestors).

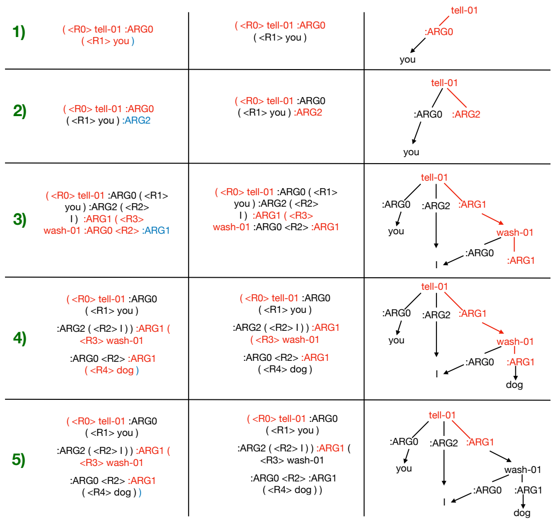
Fortunately, the DFS linearization uses several special tokens to denote the graph structure. We can rely on two special tokens to find the ancestors of a token: relation tokens (e.g. :ARG0) and the parentheses. The basic idea is: we maintain an ancestor stack for the token that will be generated, and adjust it according to the generated token. If a relation token is generated, we know that the previous siblings have been completely explored, so we will remove all the tokens of that sibling from the ancestor stack. If a right parenthesis is generated, we know that a token has been explored and we should return to its parent token, so we will remove it and all its descendants from the ancestor stacks. We always add the generated token (except the right parenthesis) into the ancestor stack after these special operations.

In Figure 5, we give an example of how to find the ancestor tokens during inference. In 1), the last token is the right parenthesis, meaning the last token *you* has been explored completely and should be removed from the ancestor token list. Therefore, we remove the tokens in the ancestor list backwards until we encounter a left parentheses. In 2), the last token is a relation token, meaning the previous sibling has been explored completely, so we remove the tokens in the ancestor list backwards until we encounter a previous relation token, then add the current relation token in the list. The steps 3), 4) and 5) are following the same rule.

3 Experiments

3.1 Setup

Dataset We use the AMR 2.0 (LDC2017T10) and AMR 3.0 (LDC2020T02) dataset. The AMR 2.0 includes 39,260 manually-created graphs, and the AMR 3.0 includes 59,255. The AMR 2.0 is a subset of AMR 3.0. Both datasets are split into training, development and test datasets.



1)
(<R0> tell-01 :ARG0 (<R1> you)
(<R0> tell-01 :ARG0 (<R1> you)
tell-01 :ARG0 you
2)
(<R0> tell-01 :ARG0 (<R1> you) :ARG2
(<R0> tell-01 :ARG0 (<R1> you) :ARG2
tell-01 :ARG0 :ARG2 you
3)
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1
tell-01 :ARG0 :ARG2 :ARG1 you wash-01 :ARG0 I :ARG1
4)
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog)
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog)
tell-01 :ARG0 :ARG2 :ARG1 you wash-01 :ARG0 I :ARG1 dog
5)
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog))
(<R0> tell-01 :ARG0 (<R1> you) :ARG2 (<R2> I)) :ARG1 (<R3> wash-01 :ARG0 <R2> :ARG1 (<R4> dog))
tell-01 :ARG0 :ARG2 :ARG1 you wash-01 :ARG0 I :ARG1 dog

Figure 5: An example of how to find ancestors during inference. The red tokens are the ancestor tokens. The left column represents the ancestor tokens for the last blue tokens. The middle column represents the change of the ancestor tokens according to the last tokens. The right column represents the ancestors in the AMR of the middle columns.

Pre-processing and Post-processing We use the same DFS-based linearization technique as Bevilacqua et al. (2021). We omit the detail here, but the reader can refer to Figure 1 as an example. In the pre-processing step, the AMR graph is linearized into a sequence, and in the post-processing step, the generated sequence is translated back to an AMR graph.

Recategorization Recategorization is a widely used technique to handle data sparsity. With recategorization, specific sub-graphs of a AMR graph (usually corresponding to special entities, like named entities, date entities, etc.) are treated as a unit and assigned to a single vertex with a new content. We experiment with a commonly-used method in AMR parsing literature (Zhang et al., 2019a,b; Zhou et al., 2020; Bevilacqua et al., 2021). The readers are referred to Zhang et al. (2019a) for further details. Notice that this method uses heuristic rules designed and optimized for AMR 2.0, and is not able to scale up to AMR 3.0 (the performance dropped substantially for AMR 3.0 with recategorization in Bevilacqua et al. (2021)). Therefore, we will not conduct the recategorization experiment on AMR 3.0.

Model and Baseline We use the model in Bevilacqua et al. (2021) as our baseline. That model was initialized by BART pretraining and fine-tuned on the AMR dataset. We will do the same thing, except that we design a different mask matrix in the Transformer Decoder layers. We will introduce these differences in detail in Section 3.2.

Training and Evaluation We use one 1080Ti GPU to fine-tune the model. Training takes about 13 hours on AMR 2.0 and 17 hours on AMR 3.0. We use the development dataset to select the best hyperparameters. At inference time, we set the beam size to 5 following common practice in neural machine translation (Yang et al., 2018).

For evaluation, we use Smatch (Cai and Knight, 2013) as the metric. For some experiments, we also report fine-grained scores on different aspects of parsing, such as wikification, concept identification, NER, and negations using the tool released by Damonte et al. (2017).

3.2 Experiments and Results

As indicated in Section 2.3, we study the effect of the hard and soft strategy. We explore the influence of these two strategies on different layers or on different heads. Due to space limitation, we only show the Smatch score of AMR 2.0 with the recategorization preprocessing, since it had the highest performance (84.5 Smatch score) as far as we know.

Once we get the best result among these setups, we will conduct experiments on AMR 2.0 and AMR 3.0 without recategorization (we have discussed why we don't conduct experiments for AMR 3.0 with recategorization before). We will also report fine-grained results for these experiments.

3.2.1 Experiments for Different Number of Heads for the Hard Strategy

In the baseline model (Bevilacqua et al., 2021), there are 16 heads in each layer. We conduct experiments with 0, 2, 4, . . . , 8, 10 heads in each layer attending to ancestors only. Note that the 0-head model equals the baseline model. We show the result in Table 2.

We can see that, up to 4 and 6 heads, the performance increases along with the number of heads increasing, showing the importance of telling the model what the ancestors are. But then, the performance decreases as the number of heads increases, showing that we cannot ignore other non-ancestor tokens, which still play important roles in the model.

Dataset	G.R.	Smatch	Unlabeled	NO WSD	Concept	SRL	Reent.	Neg.	NER	wiki
AMR 2.0 (baseline)	✓	84.5	86.7	84.9	89.6	79.7	72.3	79.9	83.7	**87.3**
AMR 2.0 (our method)	✓	**85.2**	**88.2**	**85.6**	**90.3**	**83.2**	**75.4**	**83.0**	**85.7**	86.4
AMR 2.0 (baseline)	×	83.8	86.1	84.4	90.2	79.6	70.8	**74.4**	90.6	**84.3**
AMR 2.0 (our method)	×	**84.8**	**88.1**	**85.3**	**90.5**	**83.4**	**75.1**	74.0	**91.8**	84.1
AMR 3.0 (baseline)	×	83.0	85.4	83.5	**89.8**	78.9	70.4	**73.0**	87.2	**82.7**
AMR 3.0 (our method)	×	**83.5**	**86.6**	**84.0**	89.5	**82.2**	**74.2**	72.6	**88.9**	81.5

Table 1: The smatch and fine grained scores of AMR 2.0 and AMR 3.0 datasets without recategorization using the optimal setup.

number of heads	Smatch
0 (baseline)	84.5
2	84.5
4	**84.9**
6	**84.9**
8	84.8
10	84.3

Table 2: The influence of different number of heads attended to the ancestors only for AMR 2.0 with recategorization

different layers	Smatch
baseline	84.5
bottom 4	84.6
Medium 4	**84.8**
top 4	84.3

Table 3: The influence of different layers attended to the ancestors only for AMR 2.0 with recategorization

3.2.2 Experiments for Different Layers for the Hard Strategy

In the baseline model (Bevilacqua et al., 2021), there are 12 layers in the Transformer decoder. Unlike the heads, the order of layers matters. The upper layers use information from the lower layers. Therefore, we conduct experiments with the bottom, the medium, and the top 4 layers attending to ancestors. The mask matrix for each head is the same within a single layer. We show the result in Table 3.

We can see that, putting the medium 4 layers focusing on the ancestors has the best performance. But when we put the top 4 layers focusing on them, the performance decreases a lot. One possible reason is that, when it comes to near the final output (the top layers), the model needs to use the information from all tokens.

3.2.3 Experiments of Soft Strategy

In this section, we will tune the mask matrix and use the soft strategy to add the ancestors information. We conduct three experiments: different parameters for every layer and head combination; different parameters for different layers only; different parameters for different heads only. We show the results in Table 4. We can see that when we only use different parameters for every head, we achieve a new state-of-the-art result.

different setups	Smatch
baseline	84.5
different parameters for layers and heads	84.8
different parameters only for layers	84.7
different parameters only for heads	**85.2**

Table 4: The influence of tuning the mask matrix for AMR 2.0 with recategorization

3.2.4 Results for Other Datasets

We have conducted different experiments for AMR 2.0 with recategorization, and we found that when we set different parameters for different heads only and tune these parameters, we get the best performance. Therefore, we apply this setup for other datasets: AMR 2.0 and AMR 3.0 without recategorization. We show the Smatch scores as well as other fine-grained scores in Table 1. The results are improved for all the datasets. The AMR 2.0 without recategorization even obtains an improvement of 1.0 Smatch point.

4 Conclusion

In this paper, we focus on the DFS linearization and introduce several strategies to add ancestor information into the model. We conduct experiments to show the improvement for both AMR 2.0 and AMR 3.0 datasets. Our method achieves new state-of-the-art performances for the AMR parsing task.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Research supported by NSF awards IIS-1813823 and CCF-1934962.

References

Ramón Fernandez Astudillo, Miguel Ballesteros, Tahira Naseem, Austin Blodgett, and Radu Florian. 2020. Transition-based parsing with stack-transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1001–1007.

Miguel Ballesteros and Yaser Al-Onaizan. 2017. AMR parsing using stack-LSTMs. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1269–1275.

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, Sofia, Bulgaria.

Michele Bevilacqua, Rexhina Blloshmi, and Roberto Navigli. 2021. One SPRING to rule them both: Symmetric AMR semantic parsing and generation without a complex pipeline. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*.

Deng Cai and Wai Lam. 2019. Core semantic first: A top-down approach for AMR parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3790–3800.

Deng Cai and Wai Lam. 2020. AMR parsing via graph-sequence iterative inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1290–1301, Online. Association for Computational Linguistics.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, pages 748–752.

Marco Damonte, Shay B. Cohen, and Giorgio Satta. 2016. An incremental parser for abstract meaning representation. *CoRR*, abs/1608.06111.

Marco Damonte, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for abstract meaning representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 536–546, Valencia, Spain.

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1426–1436, Baltimore, Maryland.

Zhijiang Guo and Wei Lu. 2018. Better transition-based AMR parsing with a refined search space. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1712–1722, Brussels, Belgium. Association for Computational Linguistics.

Hardy Hardy and Andreas Vlachos. 2018. Guided neural language generation for abstractive summarization using abstract meaning representation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 768–773.

Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. 2017. Neural AMR: Sequence-to-sequence models for parsing and generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 146–157, Vancouver, Canada. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880.

Fei Liu, Jeffrey Flanigan, Sam Thomson, Norman Sadeh, and Noah A. Smith. 2015. Toward abstractive summarization using semantic representations. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1077–1086.

Chunchuan Lyu and Ivan Titov. 2018. AMR parsing as graph prediction with latent alignment. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 397–407. Association for Computational Linguistics.

Arindam Mitra and Chitta Baral. 2016. Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In *AAAI*, pages 2779–2785.

Tahira Naseem, Abhishek Shah, Hui Wan, Radu Florian, Salim Roukos, and Miguel Ballesteros. 2019. Rewarding Smatch: Transition-based AMR parsing with reinforcement learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4586–4592.

Xiaochang Peng, Linfeng Song, Daniel Gildea, and Giorgio Satta. 2018. Sequence-to-sequence models for cache transition systems. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL-18)*, pages 1842–1852.

Xiaochang Peng, Chuan Wang, Daniel Gildea, and Nianwen Xue. 2017. Addressing the data sparsity issue in neural AMR parsing. In *Proceedings of the European Chapter of the ACL (EACL-17)*.

Rik van Noord and Johan Bos. 2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *Computational Linguistics in the Netherlands Journal*, 7:93–108.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008.

Chuan Wang and Nianwen Xue. 2017. Getting the most out of AMR parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1257–1268, Copenhagen, Denmark. Association for Computational Linguistics.

Tianming Wang, Xiaojun Wan, and Hanqi Jin. 2020. AMR-to-text generation with Graph Transformer. *Transactions of the Association for Computational Linguistics*, 8:19–33.

Dongqin Xu, Junhui Li, Muhua Zhu, Min Zhang, and Guodong Zhou. 2020. Improving AMR parsing with sequence-to-sequence pre-training. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2501–2511.

Yilin Yang, Liang Huang, and Mingbo Ma. 2018. Breaking the beam search curse: A study of (re-) scoring methods and stopping criteria for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3054–3059.

Shaowei Yao, Tianming Wang, and Xiaojun Wan. 2020. Heterogeneous graph transformer for graph-to-sequence learning. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL-20)*, pages 7145–7154.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019a. AMR parsing as sequence-to-graph transduction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 80–94, Florence, Italy.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019b. Broad-coverage semantic parsing as transduction. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3786–3798, Hong Kong, China. Association for Computational Linguistics.

Jiawei Zhou, Tahira Naseem, Ramón Fernandez Astudillo, and Radu Florian. 2021. AMR parsing with action-pointer transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5585–5598.

Qiji Zhou, Yue Zhang, Donghong Ji, and Hao Tang. 2020. AMR parsing with latent structural information. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4306–4319, Online. Association for Computational Linguistics.

Jie Zhu, Junhui Li, Muhua Zhu, Longhua Qian, Min Zhang, and Guodong Zhou. 2019. Modeling graph structure in Transformer for better AMR-to-text generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP-19)*, pages 5462–5471.

A Hyperparameters and Training Details

We use cross-entropy loss and RAdam optimizer during the training. We use Cosine learning rate scheduler with about 1000 warm-up steps and 20000 maximum steps. The selected value of the learning rate is 3×10^{-5}. There are around 80 sentences in each batch. We set the weight decay rate of 0.004. In order to prevent over-fitting, we use Dropout with probability 0.25, as well as label smoothing with value 0.1. To select the best model checkpoint, we use the development dataset and search for the model with the best Smatch score.