Penman: An Open-Source Library and Tool for AMR Graphs

Michael Wayne Goodman
Nanyang Technological University
Singapore
goodmami@uw.edu

Abstract

Abstract Meaning Representation (AMR) (Banarescu et al., 2013) is a framework for semantic dependencies that encodes its rooted and directed acyclic graphs in a format called PENMAN notation. The format is simple enough that users of AMR data often write small scripts or libraries for parsing it into an internal graph representation, but there is enough complexity that these users could benefit from a more sophisticated and well-tested solution. The open-source Python library *Penman* provides a robust parser, functions for graph inspection and manipulation, and functions for formatting graphs into PENMAN notation. Many functions are also available in a command-line tool, thus extending its utility to non-Python setups.

1 Introduction

Abstract Meaning Representation (AMR; Banarescu et al., 2013) is a framework for encoding English language[1] meaning as structural-semantic graphs using a fork of Propbank (Kingsbury and Palmer, 2002; O'Gorman et al., 2018) for its semantic frames with additional AMR-specific roles. The graphs are connected, directed, with node and edge labels, and may have multiple roots but always have exactly one distinguished top node. AMR corpora, such as the recent AMR Annotation Release 3.0 (LDC2020T02),[2] encode the graphs in a format called PENMAN notation (Matthiessen and Bateman, 1991). PENMAN notation is a text stream and is thus linear, but it first uses bracketing to capture a spanning tree over the graph, then inverted edge labels and references to node IDs to capture re-entrancies. Proper interpretation of the "pure" graph therefore requires the deinversion of inverted edges and the resolution of node IDs. Some tools that work with AMR use the interpreted pure graph (Cai and Knight, 2013; Song and Gildea, 2019; Chiang et al., 2013), but many others work at the tree level for surface alignment (Flanigan et al., 2014), for transformations from syntax trees (Wang et al., 2015), or to make use of tree-based algorithms (Pust et al., 2015; Takase et al., 2016). Others, particularly sequential neural systems (Konstas et al., 2017; van Noord and Bos, 2017), use the linear form directly.

Furthermore, while AMRs ostensibly describe semantic graphs abstracted away from any particular sentence's surface form, human annotators tend to "leak information" (Konstas et al., 2017) about the source sentence. This means that an annotator might be expected to produce the AMR in Fig. 1 for sentence (1), but then swap the relative order of the adjunct relations `:location` and `:time` for (2).[3] Van Noord and Bos (2017) embraced these biases and intentionally reordered relations, even frame arguments such as `:ARG0` and `:ARG1`, by their surface alignments, leading to a boost in their evaluation scores.

(1) I swam in the pool today.

(2) Today I swam in the pool.

```
(s / swim-01
  :ARG0 (i / i)
  :location (p / pool)
  :time (t / today))
```

Figure 1: An AMR for (1) or (2)

As illustrated above, work involving AMR may use the PENMAN string, the tree structure, or the

[1]Variations exist for other languages (e.g., Li et al., 2016; Cabezudo and Pardo, 2019), but AMR is primarily English and is not an interlingua (Xue et al., 2014).

[2]https://catalog.ldc.upenn.edu/LDC2020T02

[3]Graphically there is no difference, and a metric like smatch (Cai and Knight, 2013) would return a perfect score when comparing the two.

pure graph, or possibly multiple representations. This paper therefore describes and demonstrates *Penman*, a Python library and command-line utility for working with AMR data at both the tree and graph levels and for encoding and decoding these structures using PENMAN notation. Converting a tree into a graph loses information that the tree implicitly encodes, so Penman introduces the *epigraph*:[4] optional information that exists on top of the graph and controls how the pure graph is expressed as a tree. Penman is freely available under a permissive open-source license at `https://github.com/goodmami/penman/`.

2 Decoding and Encoding Graphs

Penman uses three-stage processes to decode PENMAN notation to a graph and to encode a graph to PENMAN, as illustrated in Fig. 2. **Parsing** is the process of getting a tree from a PENMAN string, and **interpretation** is getting a graph from a tree, while **decoding** is the whole string-to-graph process. Going the other way, **configuration** is the process of getting a tree from a graph and **formatting** is getting a string from a tree, while **encoding** is the whole graph-to-string process. Splitting the decoding and encoding processes into two steps each allows one to work with AMR data at any stage. The variant of PENMAN notation used by Penman is described in §2.1. The tree, graph, and epigraph data structures are described in §2.2. Getting a tree from a string (and vice-versa) depends only on understanding PENMAN notation, but getting a graph from a tree (and vice-versa) requires an understanding of the semantic model. Semantic models are described in §2.4.

2.1 PENMAN Notation

The Penman project uses a less-strict variant of PENMAN notation than is used by AMR in order to robustly handle some kinds of erroneous output by AMR parsers. The syntactic and lexical rules for PENMAN notation in PEG syntax[5] are shown in Fig. 3. Optional whitespace (not shown) is allowed around expressions in the syntactic rules.

In AMR, the `Concept` expression on `Node`, the `Atom` expression on `Concept`, and the `(Node / Atom)` expression on `Reln` are obligatory, but they are optional for Penman and will get a null value when missing. Also in AMR, the initial `Symbol` on `Node` may be further constrained with a specific `Variable` pattern for node identifiers and the `Symbol` in `Atom` would become a choice: `Variable / Symbol`. How Penman handles variables is discussed in §2.2.

AMR corpora conventionally use blank lines to delineate multiple graphs, but Penman relies on bracketing instead and whitespace is not significant. Penman also parses comments (not described in Fig. 3), which are lines prefixed with # characters, and extracts metadata where keys are tokens prefixed with two colons (e.g., `::id`) and values are anything after the key until the next key or a newline.

2.2 Decoding: Trees, Graphs, and Epigraphs

In Penman, a **tree** data structure is a $\langle n, B\rangle$ tuple where n is the node's identifier (variable) and B is a list of branches. Each branch is a $\langle l, b\rangle$ tuple where l is a branch label (a possibly inverted role) and b is a (sub)tree or an atom. The first branch on B is the node's concept, thus a tree is a near-direct conversion of the `Node` rule in Fig. 3 where B is the concatenation of `Concept` and `Reln`. The tree corresponding to the AMR in Fig. 2 is shown in Fig. 4.

A **graph** is a tuple $\langle v, T\rangle$ where v is the top variable and T is a flat list of triples. For each triple $\langle s, r, t\rangle$, the source s is always the head variable of a dependency, r is the normalized role, and t is the dependent. When interpreting a triple from a tree branch, n becomes s and t comes from b unless the branch label l is deinverted according to the semantic model (described in §2.4) to produce r, in which case s and t are swapped. In the graph, t is designated a variable if it appears as the source of any other triple; otherwise it is an atom. Triples where t is a variable are called **edge relations**. If t is an atom and r is the special role `:instance`, then t is the node's concept and the triple is an **instance relation**. All other triples are **attribute relations**. Fig. 5 shows the graph corresponding to the AMR in Fig. 2.

Conversion from a PENMAN string to a tree is straightforward: the only information lost in parsing is formatting details like the amount of whitespace. The interpretation of a graph from a tree, however, loses information about the specific tree configuration for the graph, as there are often many possible configurations for the same graph.

[4] A different sense than for an inscription on a building or a short passage at the start of a book.

[5] See `https://bford.info/packrat/`

$$\begin{tikzcd}
\text{PENMAN} \arrow[r, shift left, "\texttt{parse}"] \arrow[rr, bend left, "\texttt{decode}"] & \text{Tree} \arrow[l, shift left, "\texttt{format}"] \arrow[r, shift left, "\texttt{interpret}"] & \text{Graph} \arrow[l, shift left, "\texttt{configure}"] \arrow[ll, bend left, "\texttt{encode}"]
\end{tikzcd}$$

```
(a / alpha
   :ARG0 (b / beta)
   :ARG0-of (g / gamma
      :ARG1 b))
```

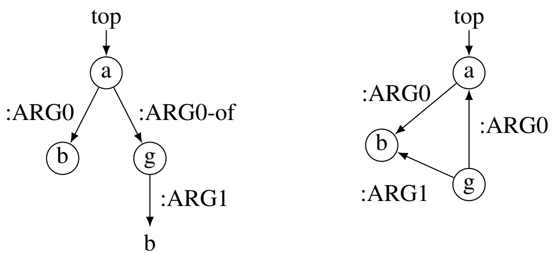

Figure 2: The three-stage decoding/encoding processes

```
# Syntactic rules
Start   <- Node
Node    <- '(' Symbol Concept? Reln* ')'
Concept <- '/' Atom?
Reln    <- Role Algn? (Node / Atom)?
Atom    <- (String / Symbol) Algn?

# Lexical rules
Symbol  <- NameChr+
Role    <- ':' NameChr*
Algn    <- '~' Prefix? Indices
Prefix  <- [a-zA-Z] '.'?
Indices <- Digit+ (',' Digit+)*
String  <- '"' (!'"' ('\\' . / .))* '"'
NameChr <- ![ \n\t\r\f\v()/:~] .
Digit   <- [0-9]
```

Figure 3: Syntactic and lexical rules of PENMAN

```
('a', [
  ('/', 'alpha'),
  (':ARG0', ('b', [
    ('/', 'beta')])),
  (':ARG0-of', ('g', [
    ('/', 'gamma'),
    (':ARG1', 'b')]))])
```

Figure 4: Tree structure for the AMR in Fig. 2

```
('a',
 [('a', ':instance', 'alpha'),
  ('a', ':ARG0', 'b'),
  ('b', ':instance', 'beta'),
  ('g', ':ARG0', 'a'),
  ('g', ':instance', 'gamma'),
  ('g', ':ARG1', 'b')])
```

Figure 5: Graph structure for the AMR in Fig. 2

Therefore, upon interpretation, Penman stores in two places the information that would be lost: in the order of triples (meaning the graph's triples are a sequence, not an unordered bag or set), and in the **epigraph**, which is a mapping of triples to lists of **epigraphical markers**. The choice of the term *epigraph* is by analogy to the *epigenome*: just as epigenetic markers control how genes are expressed in an organism, epigraphical markers control how triples are expressed in a tree. In interpreting a graph from a tree, when a branch's target is another subtree (e.g., when `(` is encountered in the string), a **Push** marker is assigned to the triple resulting from the branch, indicating that that triple pushed a new node context onto a stack representing the tree structure. The final triple resulting from branches in the subtree, even considering further nested subtrees (e.g., at the point where `)` is encountered in the string), gets a **Pop** marker indicating the end of the node context. In addition to tree-layout markers, the epigraph is also where surface alignment information is stored, as these alignments are not part of the pure graph. Fig. 6 shows the epigraph for the AMR in Fig. 2.

```
{
('a', ':instance', 'alpha'):[],
('a', ':ARG0', 'b'):         [Push('b')],
('b', ':instance', 'beta'): [Pop],
('g', ':ARG0', 'a'):         [Push('g')],
('g', ':instance', 'gamma'):[],
('g', ':ARG1', 'b'):         [Pop]
}
```

Figure 6: Epigraph structure for the AMR in Fig. 2

2.3 Encoding: No Surprises

When configuring a tree from a graph, the epigraph is used to control where triples occur in the tree. If at each step the layout markers in the epigraph allow the configuration process to navigate a tree with no surprises (that is, when the source or target of each triple is the current node on a node-context stack), then it will produce the same tree that was decoded to get the graph.[6] Otherwise, such as when a graph is modified or constructed without an epigraph, the algorithm will switch to another procedure that repeatedly passes over the list of remaining triples and configures those whose source or target are already in the tree under construction. If no triples are inserted in a pass, the remaining triples are discarded and a warning is logged that the graph is disconnected. The semantic model is used to properly configure inverted branches as necessary.

Once a tree is configured, formatting it to a string is simple, and users may customize the formatter to adjust the amount of whitespace used. The default indentation width is an adaptive mode that indents based on the initial column of the current node context; otherwise an explicit width is multiplied by the nesting level, or a user may select to print the whole AMR on one line. Another customization option is a "compact" mode which joins any attribute relations, but not edges, that immediately follow the concept onto the same line as the concept.

2.4 Semantic Models

In order to interpret a tree into a graph, a semantic model is used to get normalized, or deinverted, triples. Penman provides a default model which only checks if the role ends in `-of` (the conventional indicator of role inversion in PENMAN notation). Ideally this would be all that is needed, but AMR defines several primary (non-inverted) roles ending in `-of`, such as `:consist-of` and `:prep-on-behalf-of`, where the inverted forms are `:consist-of-of` and `:prep-on-behalf-of-of`, respectively. The model therefore first checks if a role is listed as a primary role; if not and if it ends in `-of`, it is inverted, otherwise it is not. When the role of a triple is deinverted, Penman also swaps its source and target so the dependency relation remains intact.

The model has other uses, such as inverting triples (useful when encoding), defining transformations as described in §3, and checking graphs for compliance with the model. In addition to the default model, Penman includes an AMR model with the roles and transformations defined in the AMR documentation.[7]

3 Graph and Tree Transformations

Goodman (2019a) described four transformations of AMR graphs and trees—namely, **role canonicalization**, **edge reification** (including dereification), **attribute reification**, and **tree structure indication**[8]—and how they could be used to improve the comparability of parser-produced AMR corpora by normalizing differences that are meaning-equivalent in AMR and by allowing for partial credit when, for example, a relation has a correct role but an incorrect target value. Penman incorporates all of those transformations but it (a) depends on the semantic model to define canonical roles and reifications, whereas Goodman 2019a used hard-coded transformations; and (b) inserts layout markers for a "no-surprises" configuration that results in the expected tree. A separately-defined model allows Penman to use the same transformation methods with different versions of AMR, for different tasks, or even with non-AMR representations, by creating different models. For the implementation details of these transformations, refer to Goodman 2019a.

In addition to those four transformations, Penman adds a few more methods. The **rearrange** method operates on a tree and sorts the order of branches by their branch labels. Besides changing the order of branches, their structure is unchanged by this method. Van Noord and Bos (2017) similarly rearranged tree branches based on surface alignments. The **reconfigure** method configures a tree from a graph after discarding the layout markers in the epigraph and sorting the triples based on their roles. Unlike the **rearrange** method, **reconfigure** affects the entire structure of the graph except for which node is the graph's top. For both of these, the sorting methods are defined by the

[6]There is currently one known situation where this is not the case: when a graph has duplicate triples with the same source, role, and target, as the epigraph cannot uniquely map the triple to its epigraphical markers. These, however, are likely bad graphs in AMR.

[7]`https://isi.edu/~ulf/amr/lib/roles.html`

[8]With the introduction of the epigraph, tree structure indication is somewhat redundant, however it differs in that the transformation puts this information in the graph triples.

model, and Penman includes three such methods: original order, random order, and canonical order. For **rearrange** there are additional sorting methods applicable to trees: alphanumeric order, attributes-first order, and inverted-last order. Since node variables in AMR are conventionally assigned in order of their appearance and the above methods can change this order, the **reset-variables** method reassigns the variables based on the new tree.

4 Use Cases

Here I describe a handful of use cases that motivate the use of Penman.

4.1 Graph Construction

Users of the Penman library can programmatically construct graphs and then encode them to PENMAN notation. Penman allows users to directly append to the list of triples and assign epigraphical markers, or to assemble small graphs and use set-union operations to combine them together. Another option is to assemble the tree directly, which may make more sense for some systems. Once the tree is configured or constructed, users can use transformations such as **rearrange** and **reset-variables** to make the PENMAN string more canonical in form. Fig. 7 illustrates using the Python API to construct and encode a graph.

```
>>> import penman
>>> g = penman.Graph(
...     [('s', ':instance', 'swim-01'),
...      ('s', ':ARG0', 'i'),
...      ('i', ':instance', 'i'),
...      ('s', ':location', 'p'),
...      ('p', ':instance', 'pool')])
>>> print(penman.encode(g))
(s / swim-01
   :ARG0 (i / i)
   :location (p / pool))
```

Figure 7: Example of using Penman's Python API for graph construction

Another possibility is for graph augmentation, where users rely on Penman to parse a string to a graph which they then modify, e.g., to add surface alignments or wiki links, then serialize to a string again. This allows them to focus on their primary task without worrying about the details of parsing and formatting.

4.2 Graph Validation

Whether one is generating AMR graphs with hand annotation or by automatic means, the end result is not guaranteed to be valid with respect to the model, so Penman offers a function to check for compliance. Currently, this check evaluates three criteria:

1. Is each role defined by the model?
2. Is the top set to a node in the graph?
3. Is the graph fully connected?

To facilitate both library and tool usage, the library function returns a dictionary mapping triples (for context) to error messages, as shown in Fig. 8, while the tool encodes the errors as metadata comments and has a nonzero exit-code on errors.

```
>>> from penman.models.amr import model
>>> g = penman.decode(
...      '(s / swim-01'
...      '    :ARG0 (i / i)'
...      '    :stroke (b / butterfly))')
>>> model.errors(g)
{('s', ':stroke', 'b'): ['invalid role']}
```

Figure 8: Example of using Penman's Python API for checking model compliance

4.3 Formatting for a Consistent Style

The official AMR corpora, such as the AMR Annotation Release 3.0, are distributed with the graphs serialized in a human-readable style that uses increasing levels of indentation to show the nesting of subgraphs. Furthermore, relations on a node appear in a canonical order depending on their roles (e.g., ARG1 appears before ARG2) or their surface alignments, where the appearance of a node roughly follows the order of corresponding words in a sentence. The **rearrange** and **reconfigure** transformations can change the order of relations in the graph to be more canonical, the **reset-variables** method can ensure variable forms are as expected, and the whitespace options of tree formatting can emulate the same indentation style as the official corpora. These features may be useful for users distributing new AMR corpora.

4.4 Normalization for Fairer Evaluation

The normalization options in §3 can be useful when evaluating the results of AMR parsing, as described in Goodman 2019a. Penman is thus well-situated as a preprocessor to an evaluation step using, e.g., smatch (Cai and Knight, 2013), SemBLEU (Song and Gildea, 2019), or SEMA (Anchiêta et al., 2019).

Fig. 9 shows the command-line tool performing role canonicalization.

```
$ echo '(c / chapter :domain-of 7)' \
> | penman --amr --canonicalize-roles
(c / chapter
   :mod 7)
```

Figure 9: Example of using Penman's command-line tool for normalization

4.5 Preprocessing for Machine Learning

Sequential neural models which use linearized AMR graphs have been popular for both parsing and generation (Barzdins and Gosko, 2016; Peng et al., 2017; Konstas et al., 2017; van Noord and Bos, 2017; Song et al., 2018; Damonte and Cohen, 2019; Zhang et al., 2019), but data sparsity is a significant issue (Peng et al., 2017). One way to address data sparsity is to remove senses on concepts (Lyu and Titov, 2018). Fig. 10 shows how the Python API can remove these senses in the tree.

```
>>> import re
>>> sense = re.compile(r'-\d+($|~)')
>>> def desense(branch):
...     role, tgt = branch
...     if role == '/':
...         tgt = sense.sub(r'\1', tgt)
...     return role, target
...
>>> t = penman.parse(
...     '(s / swim-01~e.1'
...     '    :ARG0 (i / i))')
>>> for _, branches in t.nodes():
...     branches[:] = map(desense,
...                       branches)
...
>>> print(penman.format(t))
(s / swim~e.1
   :ARG0 (i / i))
```

Figure 10: Example of using Penman's Python API to remove concept senses

Other techniques include, but are not limited to, normalizing linear forms, as discussed in §4.4; rearranging graphs with alignments to match the input string (van Noord and Bos, 2017); or randomizing branch orders to avoid overfitting to annotator biases, as suggested by (Konstas et al., 2017). Penman supports all these use cases via commands, as in Fig. 9, without any coding required.

5 Applicability beyond AMR

This paper has described PENMAN as a notation for encoding AMR graphs, but it is also applicable to other dependency graphs that share the same constraints (e.g., connected, directed). PENMAN notation can encode Dependency Minimal Recursion Semantics (DMRS; Copestake, 2009; Copestake et al., 2016), such as for learning graph-to-graph machine translation rules (Goodman, 2018) and neural generation (Hajdik et al., 2019), and it can encode Elementary Dependency Structures (EDS; Oepen et al., 2004; Oepen and Lønning, 2006), as shown in Fig. 11 using PyDelphin (Goodman, 2019b) for conversion. It is also useful for extensions of AMR, such as Uniform Meaning Representation (UMR; Pustejovsky et al., 2019).

```
$ echo '{e: x:pron[]
>           _1:pronoun_q[BV x]
>           e:_swim_v_1[ARG1 x]}' \
>   | delphin convert --from eds \
>                     --to eds-penman \
>                     --indent 3
(e / _swim_v_1
   :ARG1 (x / pron
      :BV-of (_1 / pronoun_q)))
```

Figure 11: Example of EDS in Penman notation

6 Conclusion

In this paper I have presented Penman, a Python library and command-line tool for working with AMR and other graphs serialized in the PENMAN format. Existing work on AMR has targeted the PENMAN string, the parsed tree, or the interpreted graph, and Penman accommodates all of these use cases by allowing users to work with the tree or graph data structures or to encode them back to strings. Transformations defined at both the graph and tree level make it applicable for pre- and post-processing steps for corpus creation, evaluation, machine learning projects, and more. Penman is available under the MIT open-source license at https://github.com/goodmami/penman. Interactive notebook demonstrations and informational videos are available at https://github.com/goodmami/penman#demo.

Acknowledgments

Thanks to the three anonymous reviewers for their helpful comments, and to the contributors and users of the Penman project for their support.

References

Rafael Torres Anchiêta, Marco Antonio Sobrevilla Cabezudo, and Thiago Alexandre Salgueiro Pardo. 2019. SEMA: an extended semantic evaluation for AMR. In *Proceedings of the 20th Computational Linguistics and Intelligent Text Processing*. Springer International Publishing.

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. Association for Computational Linguistics.

Guntis Barzdins and Didzis Gosko. 2016. RIGA at SemEval-2016 task 8: Impact of Smatch extensions and character-level neural translation on AMR parsing accuracy. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1143–1147, San Diego, California. Association for Computational Linguistics.

Marco Antonio Sobrevilla Cabezudo and Thiago Pardo. 2019. Towards a general Abstract Meaning Representation corpus for Brazilian Portuguese. In *Proceedings of the 13th Linguistic Annotation Workshop*, pages 236–244.

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 (Volume 2: Short Papers)*, volume 2, pages 748–752.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 924–932.

Ann Copestake. 2009. **Invited Talk:** slacker semantics: Why superficiality, dependency and avoidance of commitment can be the right way to go. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 1–9, Athens, Greece. Association for Computational Linguistics.

Ann Copestake, Guy Emerson, Michael Wayne Goodman, Matic Horvat, Alexander Kuhnle, and Ewa MuszyÅĎska. 2016. Resources for building applications with Dependency Minimal Recursion Semantics. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France. European Language Resources Association (ELRA).

Marco Damonte and Shay B. Cohen. 2019. Structural neural encoders for AMR-to-text generation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3649–3658, Minneapolis, Minnesota. Association for Computational Linguistics.

Jeffrey Flanigan, Sam Thomson, Jaime G 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 (Volume 1: Long Papers)*, pages 1426–1436.

Michael Wayne Goodman. 2018. *Semantic Operations for Transfer-based Machine Translation*. Ph.D. thesis, University of Washington, Seattle.

Michael Wayne Goodman. 2019a. AMR normalization for fairer evaluation. In *Proceedings of the 33rd Pacific Asia Conference on Language, Information, and Computation*, Hakodate.

Michael Wayne Goodman. 2019b. A Python library for deep linguistic resources. In *2019 Pacific Neighborhood Consortium Annual Conference and Joint Meetings (PNC)*, Singapore.

Valerie Hajdik, Jan Buys, Michael Wayne Goodman, and Emily M. Bender. 2019. Neural text generation from rich semantic representations. In *Proceedings of the 2019 Conference on the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT))*, Minneapolis, Minnesota.

Paul Kingsbury and Martha Palmer. 2002. From Treebank to Propbank. In *LREC*, pages 1989–1993. Citeseer.

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.

Bin Li, Yuan Wen, Weiguang Qu, Lijun Bu, and Nianwen Xue. 2016. Annotating the Little Prince with Chinese AMRs. In *Proceedings of the 10th Linguistic Annotation Workshop held in conjunction with ACL 2016 (LAW-X 2016)*, pages 7–15.

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.

Christian Matthiessen and John A Bateman. 1991. *Text generation and systemic-functional linguistics: experiences from English and Japanese*. Pinter Publishers.

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.

Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D Manning. 2004. LinGO Redwoods. *Research on Language and Computation*, 2(4):575–596.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based MRS banking. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*, pages 1250–1255.

Tim O'Gorman, Sameer Pradhan, Martha Palmer, Julia Bonn, Katie Conger, and James Gung. 2018. The new Propbank: Aligning Propbank with AMR through POS unification. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

Xiaochang Peng, Chuan Wang, Daniel Gildea, and Nianwen Xue. 2017. Addressing the data sparsity issue in neural AMR parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 366–375, Valencia, Spain. Association for Computational Linguistics.

Michael Pust, Ulf Hermjakob, Kevin Knight, Daniel Marcu, and Jonathan May. 2015. Parsing English into abstract meaning representation using syntax-based machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1143–1154.

James Pustejovsky, Ken Lai, and Nianwen Xue. 2019. Modeling quantification and scope in abstract meaning representations. In *Proceedings of the First International Workshop on Designing Meaning Representations*, pages 28–33, Florence, Italy. Association for Computational Linguistics.

Linfeng Song and Daniel Gildea. 2019. SemBleu: A robust metric for AMR parsing evaluation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4547–4552.

Linfeng Song, Yue Zhang, Zhiguo Wang, and Daniel Gildea. 2018. A graph-to-sequence model for AMR-to-text generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1616–1626, Melbourne, Australia. Association for Computational Linguistics.

Sho Takase, Jun Suzuki, Naoaki Okazaki, Tsutomu Hirao, and Masaaki Nagata. 2016. Neural headline generation on abstract meaning representation. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 1054–1059.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015. A transition-based algorithm for AMR parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 366–375, Denver, Colorado. Association for Computational Linguistics.

Nianwen Xue, Ondrej Bojar, Jan Hajic, Martha Palmer, Zdenka Uresova, and Xiuhong Zhang. 2014. Not an interlingua, but close: Comparison of English AMRs to Chinese and Czech. In *LREC*, volume 14, pages 1765–1772. Reykjavik, Iceland.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019. 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. Association for Computational Linguistics.