AMR Parsing using Stack-LSTMs

Miguel Ballesteros **Yaser Al-Onaizan**
IBM T.J Watson Research Center,
1101 Kitchawan Road, Route 134 Yorktown Heights, NY 10598. U.S
miguel.ballesteros@ibm.com, onaizan@us.ibm.com

Abstract

We present a transition-based AMR parser that directly generates AMR parses from plain text. We use Stack-LSTMs to represent our parser state and make decisions greedily. In our experiments, we show that our parser achieves very competitive scores on English using only AMR training data. Adding additional information, such as POS tags and dependency trees, improves the results further.

1 Introduction

Transition-based algorithms for natural language parsing (Yamada and Matsumoto, 2003; Nivre, 2003, 2004, 2008) are formulated as a series of decisions that read words from a buffer and incrementally combine them to form syntactic structures in a stack. Apart from dependency parsing, these models, also known as shift-reduce algorithms, have been successfully applied to tasks like phrase-structure parsing (Zhang and Clark, 2011; Dyer et al., 2016), named entity recognition (Lample et al., 2016), CCG parsing (Misra and Artzi, 2016) joint syntactic and semantic parsing (Henderson et al., 2013; Swayamdipta et al., 2016) and even abstract-meaning representation parsing (Wang et al., 2015b,a; Damonte et al., 2016).

AMR parsing requires solving several natural language processing tasks; mainly named entity recognition, word sense disambiguation and joint syntactic and semantic role labeling.[1] Given the difficulty of building an end-to-end system, most prior work is based on pipelines or heavily dependent on precalculated features (Flanigan et al., 2014; Zhou et al., 2016; Werling et al., 2015; Wang et al., 2015b, inter-alia).

Inspired by Wang et al. (2015b,a); Goodman et al. (2016); Damonte et al. (2016) and Dyer et al. (2015), we present a shift-reduce algorithm that produces AMR graphs directly from plain text. Wang et al. (2015b,a); Zhou et al. (2016); Goodman et al. (2016) presented transition-based tree-to-graph transducers that traverse a dependency tree and transforms it to an AMR graph. Damonte et al. (2016)'s input is a sentence and it is therefore more similar (with a different parsing algorithm) to our approach, but their parser relies on external tools, such as dependency parsing, semantic role labeling or named entity recognition.

The input of our parser is plain text sentences and, through rich word representations, it predicts all actions (in a single algorithm) needed to generate an AMR graph representation for an input sentence; it handles the detection and annotation of named entities, word sense disambiguation and it makes connections between the nodes detected towards building a predicate argument structure. Even though the system that runs with just words is very competitive, we further improve the results incorporating POS tags and dependency trees into our model.

Stack-LSTMs[2] have proven to be useful in tasks related to syntactic and semantic parsing (Dyer et al., 2015, 2016; Swayamdipta et al., 2016) and named entity recognition (Lample et al., 2016). In this paper, we demonstrate that they can be effectively used for AMR parsing as well.

2 Parsing Algorithm

Our parsing algorithm makes use of a STACK (that stores AMR nodes and/or words) and a BUFFER that contains the words that have yet to be processed. The parsing algorithm is inspired from

[1] Check (Banarescu et al., 2013) for a complete description of AMR graphs.

[2] We use the dynamic framework of Neubig et al. (2017) to implement our parser.

Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 1278–1284
Copenhagen, Denmark, September 7–11, 2017.

the semantic actions presented by Henderson et al. (2013), the transition-based NER algorithm by Lample et al. (2016) and the arc-standard algorithm (Nivre, 2004). As in (Ballesteros and Nivre, 2013) the buffer starts with the root symbol at the end of the sequence. Figure 2 shows a running example. The transition inventory is the following:

- SHIFT: pops the front of the BUFFER and push it to the STACK.
- CONFIRM: calls a subroutine that predicts the AMR node corresponding to the top of the STACK. It then pops the word from the STACK and pushes the AMR node to the STACK. An example is the prediction of a propbank sense: From *occured* **to** *occur-01*.
- REDUCE: pops the top of the STACK. It occurs when the word/node at the top of the stack is complete (no more actions can be applied to it). Note that it can also be applied to words that do not appear in the final output graph, and thus they are directly discarded.
- MERGE: pops the two nodes at the top of the STACK and then it merges them, it then pushes the resulting node to the top of STACK. Note that this can be applied recursively. This action serves to get multiword named entities (e.g. *New York City*).
- ENTITY(label): labels the node at the top of the STACK with an entity label. This action serves to label named entities, such as *New York City* or *Madrid* and it is normally run after MERGE when it is a multi-word named entity, or after SHIFT if it is a single-word named entity.
- DEPENDENT(label,node): creates a new node in the AMR graph that is dependent on the node at the top of the STACK. An example is the introduction of a negative *polarity* to a given node: From *illegal* **to** (*legal, polarity* -).
- LA(label) and RA(label): create a left/right arc with the top two nodes at the top of the STACK. They keep both the head and the dependent in the stack to allow reentrancies (multiple incoming edges). The head is now a composition of the head and the dependent. They are enriched with the AMR label.
- SWAP: pops the two top items at the top of the STACK, pushes the second node to the front of the BUFFER, and pushes the first one back into the STACK. This action allows non-projective arcs as in (Nivre, 2009) but it also helps to introduce reentrancies. At oracle time, SWAP is produced when the word at the top of the stack is blocking actions that may happen between the second element at the top of the stack and any of the words in the buffer.

Figure 1 shows the parser actions and the effect on the parser state (contents of the stack, buffer) and how the graph is changed after applying the actions.

We implemented an oracle that produces the sequence of actions that leads to the gold (or close to gold) AMR graph. In order to map words in the sentences to nodes in the AMR graph we need to align them. We use the JAMR aligner provided by Flanigan et al. (2014).[3] It is important to mention that even though the aligner is quite accurate, it is not perfect, producing a F1 score of around 0.90. This means that most sentences have at least one alignment error which implies that our oracle is not capable of perfectly reproducing all AMR graphs. This has a direct impact on the accuracy of the parser described in the next section since it is trained on sequences of actions that are not perfect. The oracle achieves 0.895 F1 Smatch score (Cai and Knight, 2013) when it is run on the development set of the LDC2014T12.

The algorithm allows a set of different constraints that varies from the basic ones (not allowing impossible actions such as SHIFT when the buffer is empty or not generating arcs when the words have not yet been CONFIRMed and thus transformed to nodes) to more complicated ones based on the propbank candidates and number of arguments. We choose to constrain the parser to the basic ones and let it learn the more complicated ones.

3 Parsing Model

In this section, we revisit Stack-LSTMs, our parsing model and our word representations.

[3] We used the latest version of the aligner (Flanigan et al., 2016)

$\textbf{Stack}_t$	$\textbf{Buffer}_t$	Action	$\textbf{Stack}_{t+1}$	$\textbf{Buffer}_{t+1}$	Graph
S	u, B	SHIFT	u, S	B	—
u, S	B	CONFIRM	n, S	B	—
u, S	B	REDUCE	S	B	—
u, v, S	B	MERGE	$(u, v), S$	B	—
u, S	B	ENTITY(l)	$(u : l), S$	B	—
u, S	B	DEPENDENT(r, d)	u, S	B	$u \xrightarrow{r} d$
u, v, S	B	RA(r)	u, v, S	B	$u \xrightarrow{r} v$
u, v, S	B	LA(r)	u, v, S	B	$u \xleftarrow{r} v$
u, v, S	B	SWAP	u, S	v, B	—

Figure 1: Parser transitions indicating the action applied to the stack and buffer and the resulting state.

Action	Stack	Buffer
INIT		It, should, be, vigorously, advocated, R
SHIFT	it	should, be, vigorously, advocated, R
CONFIRM	it	should, be, vigorously, advocated, R
SHIFT	should, it	be, vigorously, advocated, , R
CONFIRM	recommend-01, it	be, vigorously, advocated, R
SWAP	recommend-01	it, be, vigorously, advocated, R
SHIFT	it, recommend-01	be, vigorously, advocated, R
SHIFT	be, it, recommend-01	vigorously, advocated, R
REDUCE	it, recommend-01	vigorously, advocated, R
SHIFT	vigorously, it, recommend-01	advocated, R
CONFIRM	vigorous, it, recommend-01	advocated, R
SWAP	vigorous, recommend-01	it, advocated, R
SWAP	vigorous	recommend-01, it, advocated, R
SHIFT	recommend-01, vigorous	it, advocated, R
SHIFT	it, recommend-01, vigorous	advocated , R
SHIFT	it, recommend-01, vigorous	advocated, R
SHIFT	advocated, it, recommend-01, vigorous	R
CONFIRM	advocate-01, it, recommend-01, vigorous	R
LA(ARG1)	advocate-01, it, recommend-01, vigorous	R
SWAP	advocate-01, recommend-01, vigorous	it R
SHIFT	it, advocate-01, recommend-01, vigorous	R
REDUCE	advocate-01, recommend-01, vigorous	R
RA(ARG1)	advocate-01, recommend-01, vigorous	R
SWAP	advocate-01, vigorous	recommend-01, R
SHIFT	recommend01, advocate-01, vigorous	R
SHIFT	R, recommend01, advocate-01, vigorous	
LA(root)	R, recommend01, advocate-01, vigorous	
REDUCE	recommend01, advocate-01, vigorous	
REDUCE	advocate-01, vigorous	
LA(manner)	advocate-01, vigorous	
REDUCE	vigorous	
REDUCE		

```
(r / recommend-01
    :ARG1 (a / advocate-01
          :ARG1 (i / it)
          :manner (v / vigorous)))
```

Figure 2: Transition sequence for the sentence *It should be vigorously advocated*. R represents the root symbol

3.1 Stack-LSTMs

The **stack LSTM** is an augmented LSTM (Hochreiter and Schmidhuber, 1997; Graves, 2013) that allows adding new inputs in the same way as LSTMs but it also provides a POP operation that moves a pointer to the previous element. The output vector of the LSTM will consider the stack pointer instead of the rightmost position of the sequence.[4]

[4]We refer interested readers to (Dyer et al., 2015) for further details.

3.2 Representing the State and Making Parsing Decisions

The state of the algorithm presented in Section 2 is represented by the contents of the STACK, BUFFER and a list with the history of actions (which are encoded as Stack-LSTMs).[5] All of this forms the vector $\mathbf{s}_t$ that represents the state which s calculated as follows:

$$\mathbf{s}_t = \max\{\mathbf{0}, \mathbf{W}[\mathbf{st}_t; \mathbf{b}_t; \mathbf{a}_t] + \mathbf{d}\},$$

where $\mathbf{W}$ is a learned parameter matrix, $\mathbf{d}$ is a bias term and $\mathbf{st}_t$, $\mathbf{b}_t$,$\mathbf{a}_t$ represent the output vector of the Stack-LSTMs at time t.

Predicting the Actions: Our model then uses the vector $\mathbf{s}_t$ for each timestep t to compute the probability of the next action as:

$$p(z_t \mid \mathbf{s}_t) = \frac{\exp\left(\mathbf{g}_{z_t}^\top \mathbf{s}_t + q_{z_t}\right)}{\sum_{z' \in \mathcal{A}} \exp\left(\mathbf{g}_{z'}^\top \mathbf{s}_t + q_{z'}\right)} \quad (1)$$

where $\mathbf{g}_z$ is a column vector representing the (output) embedding of the action z, and q_z is a bias term for action z. The set $\mathcal{A}$ represents the actions listed in Section 2. Note that due to parsing constraints the set of possible actions may vary. The total number of actions (in the LDC2014T12 dataset) is 478; note that they include all possible labels (in the case of LA and RA) and the different dependent nodes for the DEPENDENT action

Predicting the Nodes: When the model selects the action CONFIRM, the model needs to decide the AMR node[6] that corresponds to the word at

[5] Word representations, input and hidden representations have 100 dimensions, action and label representations are of size 20.

[6]When the word at the top of stack is an out of vocabulary word, the system directly outputs the word itself as AMR node.

the top of the STACK, by using $\mathbf{s}_t$, as follows:

$$p(e_t \mid \mathbf{s}_t) = \frac{\exp\left(\mathbf{g}_{e_t}^\top \mathbf{s}_t + q_{e_t}\right)}{\sum_{e' \in \mathcal{N}} \exp\left(\mathbf{g}_{e'}^\top \mathbf{s}_t + q_{e'}\right)} \quad (2)$$

where $\mathcal{N}$ is the set of possible candidate nodes for the word at the top of the STACK. $\mathbf{g}_e$ is a column vector representing the (output) embedding of the node e, and q_e is a bias term for the node e. It is important to mention that this implies finding a propbank sense or a lemma. For that, we rely entirely on the AMR training set instead of using additional resources.

Given that the system runs two softmax operations, one to predict the action to take and the second one to predict the corresponding AMR node, and they both share LSTMs to make predictions, we include an additional layer with a *tanh* nonlinearity after $\mathbf{s}_t$ for each softmax.

3.3 Word Representations

We use character-based representations of words using bidirectional LSTMs (Ling et al., 2015b; Ballesteros et al., 2015). They learn representations for words that are orthographically similar. Note that they are updated with the updates to the model. Ballesteros et al. (2015) and Lample et al. (2016) demonstrated that it is possible to achieve high results in syntactic parsing and named entity recognition by just using character-based word representations (not even POS tags, in fact, in some cases the results with just character-based representations outperform those that used explicit POS tags since they provide similar vectors for words with similar/same morphosyntactic tag (Ballesteros et al., 2015)); in this paper we show a similar result given that both syntactic parsing and named-entity recognition play a central role in AMR parsing.

These are concatenated with pretrained word embeddings. We use a variant of the skip n-gram model provided by Ling et al. (2015a) with the LDC English Gigaword corpus (version 5). These embeddings encode the syntactic behavior of the words (see (Ling et al., 2015a)).

More formally, to represent each input token, we concatenate two vectors: a learned character-based representation ($\tilde{\mathbf{w}}_{\text{C}}$); and a fixed vector representation from a neural language model ($\tilde{\mathbf{w}}_{\text{LM}}$). A linear map ($\mathbf{V}$) is applied to the resulting vector and passed through a component-wise ReLU,

$$\mathbf{x} = \max\left\{\mathbf{0}, \mathbf{V}[\tilde{\mathbf{w}}_{\text{C}}; \tilde{\mathbf{w}}_{\text{LM}}] + \mathbf{b}\right\},$$

where $\mathbf{V}$ is a learned parameter matrix, b is a bias term and $\mathbf{w}_{\text{C}}$ is the character-based learned representation for each word, $\tilde{\mathbf{w}}_{\text{LM}}$ is the pretrained word representation.

3.4 POS Tagging and Dependency Parsing

We may include preprocessed POS tags or dependency parses to incorporate more information into our model. For the POS tags we use the Stanford tagger (Toutanova et al., 2003) while we use the Dyer et al. (2015)'s Stack-LSTM parser trained on the English CoNLL 2009 dataset (Hajič et al., 2009) to get the dependencies.

POS tags: The POS tags are preprocessed and a learned representation $\mathbf{tag}$ is concatenated with the word representations. This is the same setting as (Dyer et al., 2015).

Dependency Trees: We use them in the same way as POS tags by concatenating a learned representation $\mathbf{dep}$ of the dependency label to the parent with the word representation. Additionally, we enrich the state representation $\mathbf{s}_t$, presented in Section 3.2. If the two words at the top of the STACK have a dependency between them, $\mathbf{s}_t$ is enriched with a learned representation that indicates that and the direction; otherwise $\mathbf{s}_t$ remains unchanged. $\mathbf{s}_t$ is calculated as follows:

$$\mathbf{s}_t = \max\left\{\mathbf{0}, \mathbf{W}[\mathbf{st}_t; \mathbf{b}_t; \mathbf{a}_t; \mathbf{dep}_t] + \mathbf{d}\right\},$$

where $\mathbf{dep}_t$ is the learned vector that represents that there is an arc between the two top words at the top of the stack.

4 Experiments and Results

We use the LDC2014T12 dataset[7] for our experiments. Table 1 shows results, including comparison with prior work that are also evaluated on the same dataset.[8]

[7] This dataset is a standard for comparison and has been used for evaluation in recent papers like (Wang et al., 2015a; Goodman et al., 2016; Zhou et al., 2016). We use the standard training/development/test split: 10,312 sentences for training, 1,368 sentences for development and 1,371 sentences held-out for testing.

[8] The first entry for Damonte et al. is calculated using a pretrained LDC2015 model, available at https://github.com/mdtux89/amr-eager, but evaluated on the LDC2014 dataset. This means that the score is not directly comparable with the rest. The second entry (0.64) for Damonte et al. is calculated by training their parser with the LDC2014 training set which makes it directly comparable with the rest of the parsers.

Model	F_1(Newswire)	F_1(ALL)
Flanigan et al. (2014)* (POS, DEP)	0.59	0.58
Flanigan et al. (2016)* (POS, DEP, NER)	0.62	0.59
Werling et al. (2015)* (POS, DEP, NER)	0.62	–
Damonte et al. (2016)[8](POS, DEP, NER, SRL)	–	0.61
Damonte et al. (2016)[8](POS, DEP, NER, SRL)	–	0.64
Artzi et al. (2015) (POS, CCG)	0.66	–
Goodman et al. (2016)* (POS, DEP, NER)	0.70	–
Zhou et al. (2016)* (POS, DEP, NER, SRL)	**0.71**	**0.66**
Pust et al. (2015) (LM, NER)	–	0.61
Pust et al. (2015) (Wordnet, LM, NER)	–	**0.66**
Wang et al. (2015b)* (POS, DEP, NER)	0.63	0.59
Wang et al. (2015a)* (POS, DEP, NER, SRL)	0.70	**0.66**
OUR PARSER (NO PRETRAINED-NO CHARS)	0.64	0.59
OUR PARSER (NO PRETRAINED-WITH CHARS)	0.66	0.61
OUR PARSER (WITH PRETRAINED-NO CHARS)	0.66	0.62
OUR PARSER	0.68	0.63
OUR PARSER (POS)	0.68	0.63
OUR PARSER (POS, DEP)	0.69	0.64

Table 1: AMR results on the LDC2014T12 dataset; Newsire section (left) and full (right). Rows labeled with OUR-PARSER show our results. POS indicates that the system uses preprocessed POS tags, DEP indicates that it uses preprocessed dependency trees, SRL indicates that it uses preprocessed semantic roles, NER indicates that it uses preprocessed named entitites. LM indicates that it uses a LM trained on AMR data and WordNet indicates that it uses WordNet to predict the concepts. Systems marked with * are pipeline systems that require a dependency parse as input. (WITH PRETRAINED-NO CHARS) shows the results of our parser without character-based representations. (NO PRETRAINED-WITH CHARS) shows results without pretrained word embeddings. (NO PRETRAINED-NO CHARS) shows results without character-based representations and without pretrained word embeddings. The rest of our results include both pretrained embeddings and character-based representations.

Our model achieves 0.68 F1 in the newswire section of the test set just by using character-based representations of words and pretrained word embeddings. All prior work uses lemmatizers, POS taggers, dependency parsers, named entity recognizers and semantic role labelers that use additional training data while we achieve competitive scores without that. Pust et al. (2015) reports 0.66 F1 in the full test by using WordNet for concept identification, but their performance drops to 0.61 without WordNet. It is worth noting that we achieved 0.64 in the same test set without WordNet. Wang et al. (2015b,a) without SRL (via Propbank) achieves only 0.63 in the newswire test set while we achieved 0.69 without SRL (and 0.68 without dependency trees).

In order to see whether pretrained word embeddings and character-based embeddings are useful we carried out an ablation study by showing the results of our parser with and without character-based representations (replaced by standard lookup table learned embeddings) and with and without pretrained word embeddings. By looking at the results of the parser without character-based embeddings but with pretrained word embeddings we observe that the character-based representation of words are useful since they help to achieve 2 points better in the Newswire dataset and 1 point more in the full test set. The parser with character-based embeddings but without pretrained word embeddings, the parser has more difficulty to learn and only achieves 0.61 in the full test set. Finally, the model that does not use neither character-based embeddings nor pretrained word embeddings is the worst achieving only 0.59 in the full test set, note that this model has no explicity way of getting any syntactic information through the word embeddings nor a smart way to handle out of vocabulary words.

All the systems marked with * require that the input is a dependency tree, which means that they solve a transduction task between a dependency tree and an AMR graph. Even though our parser starts from plain text sentences when we incorporate more information into our model, we achieve further improvements. POS tags provide small improvements (0.6801 without POS tags vs 0.6822 for the model that runs with POS tags). Dependency trees help a bit more achieving 0.6920.

5 Conclusions and Future Work

We present a new transition-based algorithm for AMR parsing and we implement it using Stack-LSTMS and a greedy decoder. We present competitive results, without any additional resources and external tools. Just by looking at the words, we achieve 0.68 F1 (and 0.69 by preprocessing dependency trees) in the standard dataset used for evaluation.

Acknowledgments

We thank Marco Damonte, Shay Cohen and Giorgio Satta for running and evaluating their parser in the LDC2014T12 dataset. We also thank Chuan Wang for useful discussions.

References

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage ccg semantic parsing with amr. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710, Lisbon, Portugal. Association for Computational Linguistics.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 349–359, Lisbon, Portugal. Association for Computational Linguistics.

Miguel Ballesteros and Joakim Nivre. 2013. Going to the roots of dependency parsing. *Computational Linguistics*, 39(1).

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*.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *ACL (2)*, pages 748–752.

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

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proc. of ACL*.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah Smith. 2016. Recurrent neural network grammars. In *Proceedings of NAACL-HLT 2016*.

Jeffrey Flanigan, Chris Dyer, Noah A Smith, and Jaime Carbonell. 2016. Cmu at semeval-2016 task 8: Graph-based amr parsing with infinite ramp loss. *Proceedings of SemEval*, pages 1202–1206.

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*, volume 1, pages 1426–1436. Association for Computational Linguistics.

James Goodman, Andreas Vlachos, and Jason Naradowsky. 2016. Noise reduction and targeted exploration in imitation learning for abstract meaning representation parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1–11, Berlin, Germany. Association for Computational Linguistics.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, CoNLL '09, pages 1–18, Stroudsburg, PA, USA. Association for Computational Linguistics.

James Henderson, Paola Merlo, Ivan Titov, and Gabriele Musillo. 2013. Multilingual joint parsing of syntactic and semantic dependencies with a latent variable model. *Computational Linguistics*, 39(4):949–998.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Guillaume Lample, Miguel Ballesteros, Kazuya Kawakami, Sandeep Subramanian, and Chris Dyer. 2016. Neural architectures for named entity recognition. In *Proceedings of NAACL-HLT 2016*.

Wang Ling, Chris Dyer, Alan Black, and Isabel Trancoso. 2015a. Two/too simple adaptations of word2vec for syntax problems. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. 2015b. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Dipendra Kumar Misra and Yoav Artzi. 2016. Neural shift-reduce ccg semantic parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1775–1786, Austin, Texas. Association for Computational Linguistics.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:4:513–553. MIT Press.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 351–359. 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 *Proc. EMNLP*, Lisbon, Portugal.

Swabha Swayamdipta, Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2016. Greedy, joint syntactic-semantic parsing with stack lstms. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016*, pages 187–197.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings NAACL*.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015a. Boosting transition-based AMR parsing with refined actions and auxiliary analyzers. In *Proc. of , ACL 2015*, pages 857–862.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015b. 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.

Keenon Werling, Gabor Angeli, and Christopher D. Manning. 2015. Robust subgraph generation improves abstract meaning representation parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 982–991, Beijing, China. Association for Computational Linguistics.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 195–206.

Yue Zhang and Stephen Clark. 2011. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151.

Junsheng Zhou, Feiyu Xu, Hans Uszkoreit, Weiguang QU, Ran Li, and Yanhui Gu. 2016. Amr parsing with an incremental joint model. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 680–689, Austin, Texas. Association for Computational Linguistics.