Robust Subgraph Generation for Abstract Meaning Representation Parsing

Abstract

Abstract Meaning Representation (AMR) is a representation for open-domain rich semantics, with enormous potential as a target for parsing. We identify that broad domain node generation is a limiting factor for AMR parsing. We propose a small set of actions to construct parts of the AMR parse from spans of tokens at test time, creating the first AMR parser that can handle words at test time that were unseen during training. We show that our set of construction actions generalize better than the previous approach, even when learned with an extremely simple classifier. We improve on published state-of-the-art AMR parsing, boosting end-to-end F1 from 0.59 to 0.62 on the LDC2013E117 and LDC2014T12 datasets.

1 Introduction

Abstract Meaning Representation (AMR) (Banarescu et al., 2013) is a rich graph-based language for expressing semantics over a broad domain. Figure 1 shows an example AMR for "he gleefully ran to his dog Rover", and we give a brief tutorial on AMR in Section 2.

AMR parsing is the task of mapping a natural language sentence into an AMR graph. AMR parsing is exciting because a practical broad-domain AMR parser could enable a new breed of natural language applications ranging from semantically aware MT to rich broad-domain QA over text-based knowledge bases. At the time of this writing AMR is the target of a multi-institution multi-year data-labeling program led by Kevin Knight, which promises to produce a corpus that is large enough to make it possible to parse to AMR well despite its many challenges.

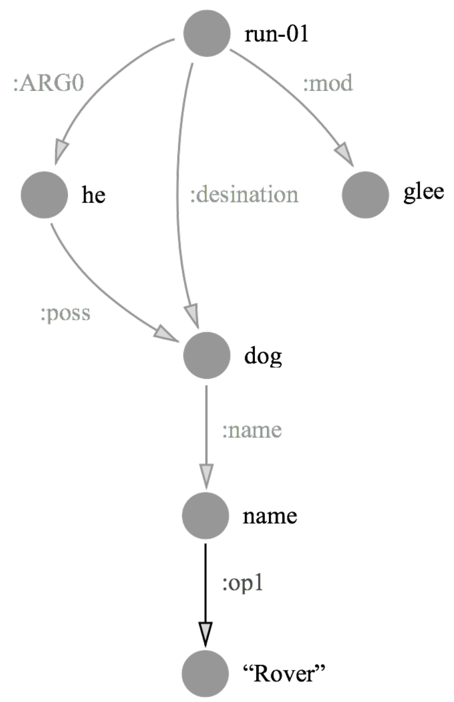

Figure 1: AMR graph for "He gleefully ran to his dog Rover". Nodes represent concepts, and arcs are relationships between concepts. The dark arc labeled "op1" is expected to be generated by NER++.

No matter how much data we are soon to have, though, AMR parsing remains a hard unsolved task, with only modest performance numbers reported so far (including the system contributed by this paper). A successful AMR parser will need to handle every SemEval task well, and conquer several additional semantic parsing challenges unique to AMR. AMR also has nasty structural properties, providing no guarantees about projectivity or even acyclicity of its graphs.

In our dicussion of the challenges inherent in AMR parsing, we follow the convention of dividing AMR parsing into two steps, first proposed by (Flanigan et al., 2014). The first step is *concept*

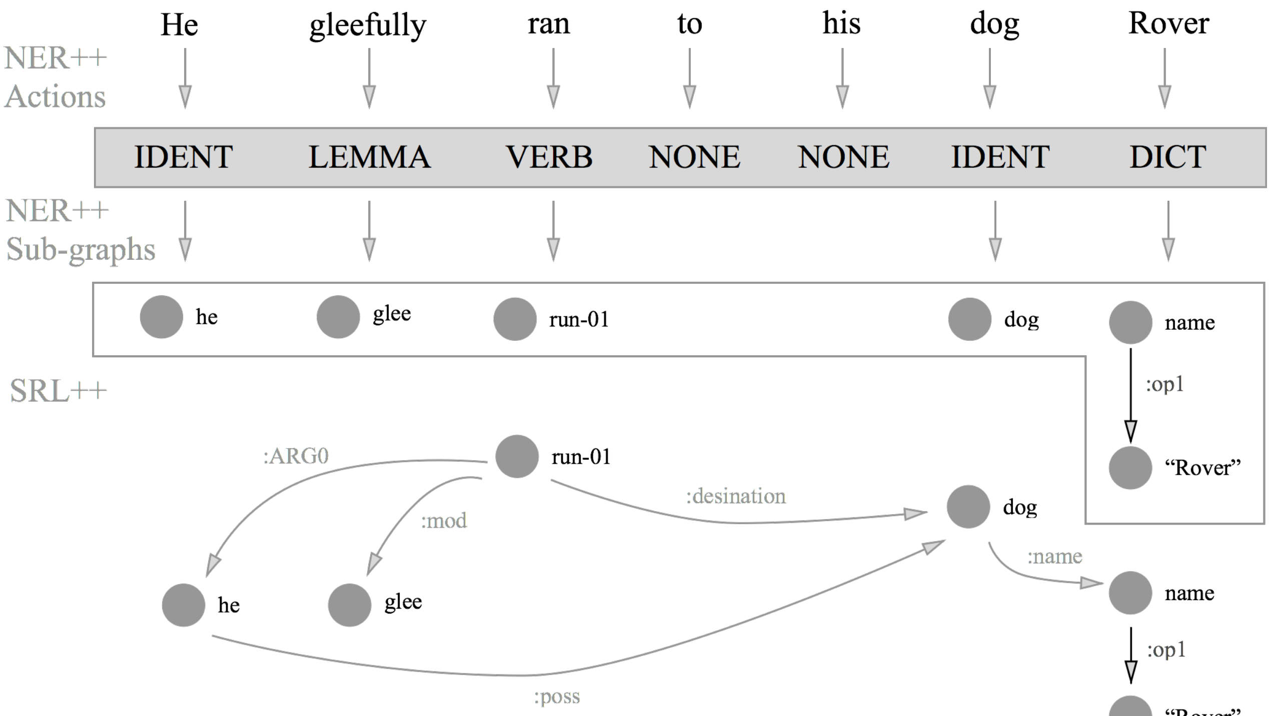

Figure 2: Derivation process for "He gleefully ran to his dog Rover". First the tokens in the sentence are labeled with derivation actions, then those actions are used to generate AMR sub-graphs, and then those sub-graphs are stitched together to form a coherent whole.

identification, which generates nodes from text, and which we'll refer to as *NER++* (Section 3.1). The second step is *relation identification*, which adds arcs to link these nodes into a fully connected AMR graph, which we'll call *SRL++* (Section 3.2).

A reasonable NLP researcher posessing a background in structure-prediction might look at AMR parsing and assume that the area deserving attention is predicting the edges in these nasty cyclic non-projective graphs (SRL++). This would be supported by a perfectly plausible analogy to existing dependency parsers. Such a researcher might spend months testing novel structure-prediction algorithms and training regimes to capture aspects of the non-projective cyclic graphs, only to find that with a careful application of MST, (Flanigan et al., 2014) has set an extremely strong baseline on the SRL++ task, and gains are negligible at best.

This is because, contrary to perfectly reasonable intuition, SRL++ is *not* the hard part of AMR parsing. The hard part of AMR parsing is NER++.

Our first piece of evidence in making this claim is a close analysis of the only published AMR parser to date, (Flanigan et al., 2014), dubbed "JAMR". When JAMR makes parses using its own NER++ and SRL++ components, it gets an end-to-end score of 0.58 F1. However, when JAMR is given a gold NER++ output, and must only perform SRL++ over given sub-graphs it scores *0.80* F1. Inter-annotator agreement on AMR is 0.83 F1 (and annotators very rarely disagree on NER++ **TODO:** verify, which amounts to disagreeing about the entities mentioned in a sentence) so this means the SRL++ component is nearly perfect, if given perfect NER++.

Upon reflection it makes sense that SRL++ within AMR is *relatively* easy given a perfect set of nodes to link together. There's a strong type-check feature for the existence and type of any arc just by looking at its end-points, and dependency features are very informative for removing any remaining ambiguity. If a system is considering how to link the node "run-01" in Figure 1, the verb-sense frame for "run-01" leaves very little entropy in terms of what we could assign as an ARG0 arc. It must be a noun, which leaves either "he" or "dog", and this is easily decided in favor of "he" by looking for an nsubj arc in the dependency parse. Given a modest amount of data, a simple parser can learn these lexical and syntactic type-check rules.

While SRL++ is hard, NER++ is extremely challenging, encompassing many SemEval tasks in a single monolith. This task was handled by a simple baseline system in (Flanigan et al., 2014), which memorized a purely lexical mapping from spans of text to AMR nodes, augmented by an NER system and time expression regex at test time. This is problematic, because fully 38% of the tokens in the test set are unobserved at training time, and the existing method has no way to handle generation for them.

The primary contribution of this paper is a method to largely delexicalize NER++ to gracefully handle unseen tokens. Choosing from among a small set of 'generative actions' our system can derive an AMR sub-graph from a span of tokens (see Figure 2). For example, we have an action *VERB* that will perform a verb-sense-disambiguation to the appropriate PropBank frame (**?**)n the source token, like the "ran" to "run-01" example in Figure 1.

We show this approach improves recall dramatically over previous approaches, and that end to end performance is improved 0.59 to 0.62 smatch when our generative actions are stitched together by the previous state of the art parser (Flanigan et al., 2014). We suspect that the modest improvement is due to the fact that our NER++ system is able to generate nodes that the SRL++ system has never seen during training time, which makes the lexicalized typecheck features useless.

2 A Crash-Course in AMR

AMR is a language for expressing semantics as a rooted, directed, and potentially cyclic graph, where nodes represent concepts and arcs are relationships between concepts. AMR is based on neo-Davidsonian semantics, (Davidson, 1967), and propositional semantics, (Parsons, 1990). AMR makes no effort to have a one-to-one correspondence between nodes in a graph and tokens in the sentence whose semantics is being represented. In fact, AMR will often expand single tokens into large sub-graph elements, or ignore tokens completely. It's important to understand that AMR represents the relationships between objects referred to by the surface text, *not* the relationships between the words themselves.

To introduce AMR and its notation, we'll unpack the translation of the sentence "he gleefully ran to his dog Rover". We show in Figure 1 the interpretation of this sentence as an AMR graph.

Note that the root node of the graph is labeled "run-01". This is the name of a verb sense definition drawn from PropBank [citation needed] for the sense of the verb "ran" in this sentence.

"run-01" has an outgoing "ARG0" arc to a node "he", with semantics (drawn from the PropBank frame) that roughly correspond to "he" being the doer of the "run-01" action. The "run-01" has an outgoing "mod" to "glee," which has the catch-all semantics that "run-01" is somehow modified by the concept "glee." "run-01" also has a "destination" arc to "dog," which preposition senses from (Srikumar, 2013) [citation needed], and means that the destination of the "run-01" action is "dog". Then we have a section of the graph that is best interpreted as a unit, where all of the children of "dog" effectively mean that "dog" has the name "Rover."

2.1 Formal task definition

Formally, given an array of tokens $S = [s_0, \ldots, s_n]$, generate a directed AMR graph G, defined as the pair $(N = [n_0, \ldots, n_k], A \in L^{k*k})$, where N is an array of AMR nodes (which doesn't have to be the same length or have a clear correspondence to S), and A is a matrix of labels L, where $A_{i,j} = l \in L$ means that an arc exists from node n_i to node n_j with label l in the parsed graph. We include the special label "NONE" in L, corresponding to no arc existing between two nodes.

2.2 AMR Subgraphs

AMR contains components that, while they may be composed of multiple nodes, can logically be considered the expression of a single concept. For the NER++ task, we would like to be able to generate these "single concept subgraphs" directly from spans of text.

AMR makes an attempt to capture some semantic meanings in words that are difficult to capture in a way that is not domain specific. For example, the token "sailor" in a sentence will evoke the concept graph representing a person who performs the action "sail-01", see Figure 3. This is difficult to model without resorting to memorization, because the etymological clues are so sparse. We note this as an area for further exploration.

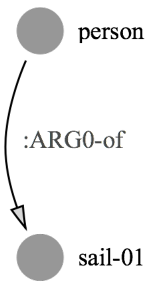

Figure 3: AMR representation of the word "sailor", which is notable for breaking the word up into a self-contained multi-node unit signifying some etymological understanding.

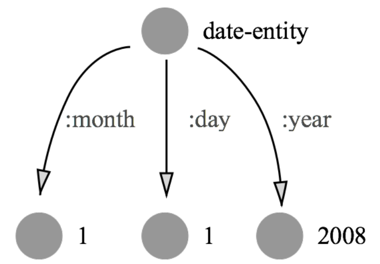

Figure 4: AMR representation of the span "January 1, 2008", an example of how AMR can represent structured data by creating additional nodes like "date-entity" to signify the presence of special structure

AMR can also capture structured data, like time expressions, see Figure 4. In dates, a "date-entity" node is created to signify that this cluster of nodes is part of a structured sub-component of an AMR graph, with specific semantics. Dates are a good example of a recurring pattern in AMR, which is to have an "artificial node" signify that all its immediate children are part of a structured piece of data, with some special interpretation. The most common example of this pattern is the "name" node, which signifies that its immediate children comprise the tokens of a name object.

3 Task decomposition

We define the terms **NER++** and **SRL++** in more detail within this section.

3.1 NER++

Parsing to the AMR representation demands a rich NER system, word sense disambiguation, number normalization, time parsing, and many semantic nominalizations and part of speech translations, and within-sentence coreference. We refer to these "low-level" AMR tasks collectively as NER++. NER++ is the sub-task of generating the best AMR sub-graphs ("sub-graph" is defined in Section 2.2) given the set of tokens S. This involves both partitioning the source text into spans that will be rendered as a single sub-graph in AMR (e.g. "run", "People's Republic of China", "January 1, 2008"), and then mapping each of those spans into a corresponding AMR sub-graph of maximum likelihood.

3.2 SRL++

Given a perfect NER++ system for an AMR parser, there remains the task of noting the verb arguments, preposition sense actionging, and doing some augmented semantic dependency parsing in order to join the disjoint NER++ output into a single AMR parse. We call this task SRL++. SRL++ is the sub-task of taking as input the disjoint sub-graphs generated by NER++, and adding the maximum likelihood set of arcs between the sub-graphs in order to have a fully connected graph.

4 Previous Work

At the time of this writing, the JAMR parser (Flanigan et al., 2014) is the only published AMR parser. The crucial insight in JAMR is that AMR parsing can be broken into two relatively distinct tasks: interpreting what entities are being referred to in the text (which we call NER++), and then discovering what relationships those entities have between one another other (which we call SRL++).

For NER++, JAMR uses a simple Viterbi sequence model to directly generate AMR-subgraphs from memorized mappings of text spans to subgraphs. Then for SRL++ JAMR uses a variation of the maximum spanning tree algorithm augmented by dual decomposition to impose linguistically motivated constraints on a maximum likelihood stitching. JAMR's SRL++ component is extremely effective, and we were unable to produce a better SRL++ system in our experiments with several other structured prediction approaches.

Semantic parsing has been explored extensively outside of AMR. **TODO:** Cite Percy, Zettlemoyer, etc

5 NER++ Method

Our approach to improving NER++ is very simple: instead of trying to pick which of thousands of AMR sub-graphs to generate from a span of text directly, we partition the AMR sub-graph space in terms of the actions needed to derive a node from its aligned token. At test time we do a sequence labeling of input tokens with these actions, and then deterministically derive the AMR sub-graphs from spans of tokens by applying the transformation decreed by their actions. This dramatically reduces sparsity, and helps improve end-to-end performance, but is most beneficial for domain transfer. We explain in Section 5.2 how exactly we manage this partition, and explain in Section 5.4 how we create training data from existing resources to train a action-type classifier. Then we setup the classifier itself in Section 5.5.

5.1 Derivation actions

We partition the AMR sub-graph space into a set of 7 actions, each corresponding to an action that will be taken by the NER++ system if a token receives this classification.

- **VERB**: Look for the most similar PropBank frame, make that the title of the corresponding node.
- **IDENTITY**: Take the lowercased version of the token to be the title of the corresponding node.
- **VALUE**: Parse the token to an integer value, and use that as the node. AMR actually does type-check, so
- **LEMMA**: Take the lemma of the token to be the title of the corresponding node.
- **NONE**: Ignore this token in the final output.
- **NAME**: Attach a created "name" node to the top of this span, but don't add an NER action type on top of the "name" node.
- **DICT**: Look up the most probable chunk associate with this lexical span. This functions as a back off if no other actions are appropriate.

5.2 Notes on the DICT action

It's not always possible to derive an AMR sub-graph directly from tokens at test time without having memorized a mapping. For example, the parse of "sailor" as "person who sails", see Figure 3, is nearly impossible without some form of memorization. That's where the **DICT** class is important.

To implement a **DICT** class, we memorize a simple mapping from spans of text, like "sailor" to their corresponding most frequently seen AMR sub-graphs in the training data, in this case Figure 3. At test time we can do a lookup in this dictionary for any element that gets labeled with a **DICT** action. Previous approaches have been the equivalent of labeling every node with the **DICT** action, so our reduction of its use is significant. This is the distribution of actions on the LDC2014T12 proxy training data, after our automatic alignment allows us to induce actions (see Section 5.4 for how this is done).

Action	# Tokens	% Total
NONE	41538	0.371
DICT	30027	0.268
IDENTITY	19034	0.170
VERB	11739	0.104
LEMMA	5029	0.045
NAME	4537	0.04
VALUE	16	0.001

Table 1: Distribution of action types in the proxy section of the LDC2014T12 dataset, generated from automatically aligned data.

Note that **DICT** counts for around 27% of the training data, meaning that more than 72% of tokens can be generated correctly by our action type classifier even if we've never seen them before, which is a huge win.

We believe that **DICT** should count for much less than 27%, and **LEMMA** should count for much more than 4%, but issues with existing lemmatizers prevent this, see our error analysis in Section 7.

5.3 Action Informativeness Hierarchy

We define the concept of "action informativeness" of an action a as the probability of deriving the correct node from a span of tokens, given that those tokens are labeled with the action a, and a is the correct action for that span of tokens.

To provide a concrete example, our dictionary lookup classifier has a test-set accuracy of 0.67. That means that the "action informativeness" of the **DICT** action is 0.67, because given that we correctly label a token as **DICT**, there is a probability of 0.67 that we correctly generate the corre-

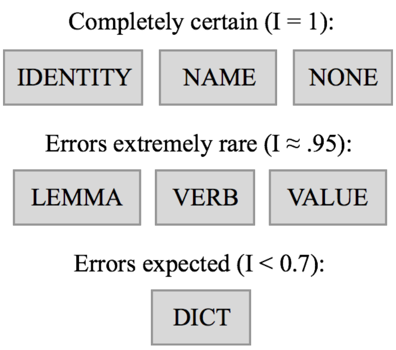

Figure 5: Informativeness hierarchy for action tags within AMR.

sponding node.

In contrast to **DICT**, correctly labeling a node as **IDENTITY**, **NAME**, and **NONE** have action informativeness of 1.0, since there is no ambiguity in the node generation once one of those actions have been selected, and we are guaranteed (probability 1.0) to generate the correct node given the correct action.

This allows us to induce an action informativeness hierarchy, with more informative actions taking precedence over less informative actions for several important tasks. We demonstrate this hierarchy in Figure 5.

5.4 Inducing Derivation actions from Training Data

Given a set of AMR training data, in the form of (graph,sentence) pairs, we first induce alignments from the graph nodes to the sentence, see Section **??**. Given an alignment, which is an annotation on the graph noting for each node N_i a token S_j that is most likely to have "generated" N_i, we can induce alignments. For concreteness, imagine the token S_j is "running", and the node N_i has the title "run-01". For each action type, we can ask whether that action type is able to take token S_j and correctly generate N_i. The two action types we find that are able to correctly generate this node are **DICT** and **VERB**. We choose the most informative action type of those available to generate the observed node. In this case, that means we choose **VERB**.

In general, our algorithm is as follows. For all S_j to which no N_i exists such that N_i aligns to S_j, assign the action **NONE** to S_j. For all pairs N_i, S_j, assign S_j the most informative action possible that could have generated N_i.

TODO: Flesh out discussion of adjacent DICT nodes

5.5 Action Classifier

We use an extremely simple max-ent classifier to make action decisions. The classifier takes as input a pair $< i, S >$, where i is the index of the token in the input sentence, and S is a sequence tokens representing the source sentence. The output of the classifier is a action T such that the likelihood with respect to the data of token i in sentence S generating a node according to the action specified by T is maximized. See Appendix A for a list of classifier features.

5.6 Test Time Behavior

At test time, given a sequence of input tokens, we do a simple classification of each token separately, to get a sequence labeling of our input tokens. Then for each token, we apply the behavior associated with the token label, and the resulting set of sub-graphs is passed on to SRL++ for linking.

6 Results

6.1 End to end results

Our end to end results are reported by plugging the output of our NER++ into the SRL++ component of JAMR (Flanigan et al., 2014), which is able to produce final AMR graphs when given a sequence of spans and their corresponding chunks. AMR parsing accuracy is measured with a metric called smatch [citation needed], which stands for "s(emantic) match". The metric is the F1 of a best-match between triples implied by the target graph, and triples in the parsed graph. We report much higher recall, and a slightly improved F1 score.

NER++	Dataset	P	R	**F1**
JAMR	LDC2014T12	**0.671**	0.532	0.59
Robust	LDC2014T12	0.639	**0.596**	**0.62**
JAMR	LDC2014E117	**0.669**	0.529	0.59
Robust	LDC2014E117	0.636	**0.601**	**0.62**

Table 2: Results on two AMR datasets.

TODO: interpretation

6.2 Component results

On our action-type sequence labeling data generated from automatic alignments on train and test splits of *LDC2014T12*, our classifier achieved a test accuracy of **0.841**.

The **DICT** action lookup table achieved an accuracy of **0.67** on the test set. This is remarkable, given that our model moves many of the difficult semantic tasks onto the **DICT** tag, and we are using no learning here beyond a simple count of observed span to sub-graph mappings.

7 Error Analysis

7.1 Weak lemmatization

The **DICT** class was intended to be used for things that a system cannot know without memorization, like the nominalization of "sailor", see Figure 3. These don't occur nearly 25% of the time in the training data. One of the reasons that the **DICT** class is so disappointingly large is that it's stealing from **LEMMA**, because AMR will aggressively normalize words and change their part of speech to a semantic neighbor. For example, 'gleefully' gets mapped to 'glee' and not 'gleeful', which is hard to do automatically with stemming rules in the general case. We leave this as a direction for future work.

7.2 DICT Classifier

The **DICT** class is surprisingly large, and our attempts to handle node generation within the class were fairly feeble.

8 Future Work

8.1 Semantically equivalent POS normalization

The benefit of this approach could be increased by having a very strong stemmer tuned to AMR parsing, which currently doesn't exist.

8.2 Etymological approach to node generation

There is an opportunity to create and test etymologico-semantic approaches to parsing words like 'sailor' that would benefit AMR parsing domain generalization tremendously. We envision a system that upon discovering the unseen noun 'arbitrageur' at training time is able to retrieve the know verb lemma 'arbitrage', and derive that the noun 'arbitrageur' probably refers to an entity that is engaged in arbitrage. AMR training data provides an opportunity to perform and measure such a task in isolation,

NER++ Features
Input token
Input token word embedding
Left token
Right token
Left bigram
Right bigram
POS
Left POS
Right POS
Left POS bigram
Right POS bigram
Token's dependency parent token
Token's dependency parent POS
Token's dependency parent arc name
Bag of outgoing dependency arcs
Number of outgoing dependency arcs
Number of outgoing dependency arcs (indicator)
Max JaroWinker to any lemma in PropBank
Closest (JaroWinkler) in PropBank
Token NER
Left NER bigram
Right NER bigram
Right NER bigram
Indicator for if token is a recognized AMR NER type
Indicator for if token is capitalized
Parent arc is prep_* or appos, and parent has NER action
Indicator for token is pronoun
Indicator for token is part of a coref chain
Indicator for token pronoun and part of a coref chain

Table 3: The features for the NER++ max-ent classifiers.

9 Appendix

Acknowledgments

The acknowledgments should go immediately before the references. Do not number the acknowledgments section. Do not include this section when submitting your paper for review.

References

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, Noah A. Smith 2014. *ACL 14*, volume 1.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider 2013. *Proc. of the Linguistic Annotation*

Workshop and Iteroperability with Discourse, volume 1.

Terence Parsons 1990 *Events in the Semantics of English: A study in subatomic semantics*. MIT Press.

Donald Davidson 1967 *The Logic of Decision and Action*, pages 81-120. Univ. of Pittsburg Press.

Nima Pourdamghani, Yang Gao, Ulf Hermjakob and Kevin Knight 2014 *Aligning English Strings with Abstract Meaning Representation Graphs* Proc. EMNLP.

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation and Compiling*, volume 1. Prentice-Hall, Englewood Cliffs, NJ.

American Psychological Association. 1983. *Publications Manual*. American Psychological Association, Washington, DC.

Association for Computing Machinery. 1983. *Computing Reviews*, 24(11):503–512.

Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. 1981. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133.

Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK.