

An Incremental Parser for Abstract Meaning Representation

Marco Damonte
School of Informatics
University of Edinburgh
m.damonte@sms.ed.ac.uk

Shay B. Cohen
School of Informatics
University of Edinburgh
scohen@inf.ed.ac.uk

Giorgio Satta
Dept. of Information Engineering
University of Padua
satta@dei.unipd.it

Abstract

Abstract Meaning Representation (AMR) is a semantic representation for natural language that embeds annotations related to traditional tasks such as named entity recognition, semantic role labeling, word sense disambiguation and co-reference resolution. We describe a transition-based parser for AMR that parses sentences left-to-right, in linear time. We further propose a test-suite that assesses specific sub-tasks that are helpful in comparing AMR parsers, and show that our parser is competitive with the state of the art on the LDC2015E86 dataset and that it outperforms state-of-the-art parsers for recovering named entities and handling polarity.

1 Introduction

Semantic parsing aims to solve the problem of canonicalizing language and representing its meaning: given an input sentence, it aims to extract a semantic representation of that sentence. Abstract meaning representation (Banarescu et al., 2013), or AMR for short, allows us to do that with the inclusion of most of the shallow-semantic natural language processing (NLP) tasks that are usually addressed separately, such as named entity recognition, semantic role labeling and co-reference resolution. AMR is partially motivated by the need to provide the NLP community with a single dataset that includes basic disambiguation information, instead of having to rely on different datasets for each disambiguation problem. The annotation process is straightforward, enabling the development of large datasets.

Several parsers for AMR have been recently developed (Flanigan et al., 2014; Wang et al., 2015a; Peng et al., 2015; Pust et al., 2015; Goodman et al., 2016; Rao et al., 2015; Vanderwende et al., 2015; Artzi et al., 2015; Barzdins and Gosko, 2016; Zhou et al., 2016). This line of research is new and current results suggest a large room for improvement. Greedy transition-based methods (Nivre, 2008) are one of the most popular choices for dependency parsing, because of their good balance between efficiency and accuracy. These methods seem promising also for AMR, due to the similarity between dependency trees and AMR structures, i.e., both representations use graphs with nodes that have lexical content and edges that represent linguistic relations.

A transition system is an abstract machine characterized by a set of configurations and transitions between them. The basic components of a configuration are a stack of partially processed words and a buffer of unseen input words. Starting from an initial configuration, the system applies transitions until a terminal configuration is reached. The sentence is scanned left to right, with linear time complexity for dependency parsing. This is made possible by the use of a greedy classifier that chooses the transition to be applied at each step.

In this paper we introduce a parser for AMR that is inspired by the ARCEAGER dependency transition system of Nivre (2004). The main difference between our system and ARCEAGER is that we need to account for the mapping from word tokens to AMR nodes, non-projectivity of AMR structures and reentrant nodes (multiple incoming edges). Our AMR parser brings closer dependency parsing and AMR parsing by showing that dependency parsing algorithms, with some modifications, can be used for AMR. Key properties

such as working left-to-right, incrementality¹ and linear complexity further strengthen its relevance.

The AMR parser of Wang et al. (2015a), called CAMR, also defines a transition system. It differs from ours because we process the sentence left-to-right while they first acquire the entire dependency tree and then process it bottom-up. More recently Zhou et al. (2016) presented a non-greedy transition system for AMR parsing, based on ARC-STANDARD (Nivre, 2004). Our transition system is also related to an adaptation of ARCEAGER for directed acyclic graphs (DAGs), introduced by Sagae and Tsujii (2008). This is also the basis for Ribeyre et al. (2015), a transition system used to parse dependency graphs. Similarly, Du et al. (2014) also address dependency graph parsing by means of transition systems. Analogously to dependency trees, dependency graphs have the property that their nodes consist of the word tokens, which is not true for AMR. As such, these transition systems are more closely related to traditional transition systems for dependency parsing.

Our contributions in this paper are as follows:

- In §3 we develop a left-to-right, linear-time transition system for AMR parsing, inspired by the ARCEAGER transition system for dependency tree parsing;
- In §5 we claim that the Smatch score (Cai and Knight, 2013) is not sufficient to evaluate AMR parsers and propose a set of metrics to alleviate this problem and better compare alternative parsers;
- In §6 we show that our algorithm is competitive with publicly available state-of-the-art parsers on several metrics.

2 Background and Notation

AMR Structures AMRs are rooted and directed graphs with node and edge labels. An annotation example for the sentence *I beg you to excuse me* is shown in Figure 1, with the AMR graph reported in Figure 2.

Concepts are represented as labeled nodes in the graph and can be either English words (e.g. *I* and *you*) or Propbank framesets (e.g. *beg-01* and *excuse-01*). Each node in the graph is assigned to

¹Strictly speaking, transition-based parsing cannot achieve full incrementality, which requires to have a single connected component at all times (Nivre, 2004).

(**b** / *beg-01*
:ARG0 (**i** / *i*
:ARG1 (**y** / *you*)
:ARG2 (**e** / *excuse-01*
:ARG0 **y**
:ARG1 **i**))

Figure 1: Annotation for the sentence “*I beg you to excuse me.*” Variables are in boldface and concepts and edge labels are in italics.

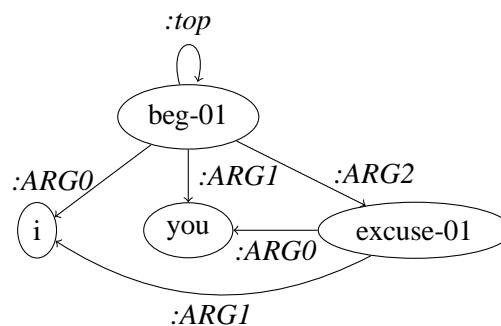


Figure 2: AMR graph representation for Figure 1.

a variable in the AMR annotation so that a variable re-used in the annotation corresponds to reentrancies (multiple incoming edges) in the graph. Relations are represented as labeled and directed edges in the graph.

Notation For most sentences in our dataset, the AMR graph is a directed acyclic graph (DAG), with a few specific cases where cycles are permitted. These cases are rare, and for the purpose of this paper, we consider AMR as DAGs.

We denote by $[n]$ the set $\{1, \dots, n\}$. We define an AMR structure as a tuple (G, x, π) , where $x = x_1 \dots x_n$ is a sentence, with each $x_i, i \in [n]$, a word token, and G is a directed graph $G = (V, E)$ with V and E the set of nodes and edges, respectively.² We assume G comes along with a node labeling function and an edge labeling function. Finally, $\pi: V \rightarrow [n]$ is a total alignment function that maps every node of the graph to an index i for the sentence x , with the meaning that node v represents (part of) the concept expressed by the word $x_{\pi(v)}$.³

²We collapse all multi-word named entities in a single token (e.g., *United Kingdom* becomes *United.Kingdom*) both in training and parsing.

³ π is a function because we do not consider co-references, which would otherwise cause a node to map to multiple indices. This is in line with current work on AMR parsing.

We note that the function π is not invertible, since it is neither injective nor surjective. For each $i \in [n]$, we let

$$\pi^{-1}(i) = \{v \mid v \in V, \pi(v) = i\}$$

be the pre-image of i under π (this set can be empty for some i), which means that we map a token in the sentence to a set of nodes in the AMR. In this way we can align each index i for x to the induced subgraph of G . More formally, we define

$$\check{\pi}(i) = (\pi^{-1}(i), E \cap (\pi^{-1}(i) \times \pi^{-1}(i))), \quad (1)$$

with the node and edge labeling functions of $\check{\pi}(i)$ inherited from G . Hence, $\check{\pi}(i)$ returns the AMR subgraph aligned with a particular token in the sentence.

2.1 Transition-Based AMR Parsing

Similarly to dependency parsing, AMR parsing is partially based on the identification of predicate-argument structures. Much of the dependency parsing literature focuses on *transition-based* dependency parsing—an approach to parsing that scans the sentence from left to right in linear time and updates an intermediate structure that eventually ends up being a dependency tree.

The two most common transition systems for greedy dependency parsing are ARCSTANDARD and ARCEAGER. With ARCSTANDARD, a stack is maintained along with a buffer on which the left-to-right scan is performed. At each step, the parser chooses to scan a word in the buffer and shift it onto the stack, or else to create an arc between the two top-most elements in the stack and pop the dependent. ARCSTANDARD parses a sentence in a pure bottom-up, left-to-right fashion (similarly to shift-reduce context-free grammar parsers), and must delay the construction of right arcs until all the dependent node has been completed. This imposes strong limitations on the degree of incrementality of the parser. The ARCEAGER system was designed to improve on ARCSTANDARD by mixing bottom up and top-down strategies. More precisely, in the ARCEAGER parser left arcs are constructed bottom-up and right arcs are constructed top-down, so that right dependents can be attached to their heads even if some of their own dependents are not identified yet. In this way arcs are constructed as soon as the head and the dependent are available in the stack.

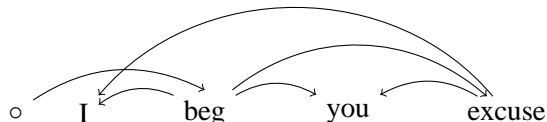


Figure 3: AMR’s edges for the sentence “I beg you to excuse me.” mapped back to the sentence, according to the alignment. \circ is a special token representing the root.

Because of the similarity of AMR structures to dependency structures, transition systems are also helpful for AMR parsing. Starting from the ARCEAGER system, we develop here a novel transition system, called AMREAGER that parses sentences into AMR structures. There are three key differences between AMRs and dependency trees that require further adjustments for dependency parsers to be used with AMRs.

Non-Projectivity A key difference between English dependency trees and AMR structures is projectivity. Dependency trees in English are usually projective, roughly meaning that there are no crossing arcs if the edges are drawn in the semi-plane above the words. While this restriction is empirically motivated in syntactic theories for English, it is no longer motivated for AMR structures.

The notion of projectivity can be generalized to AMR graphs as follows. The intuition is that we can use the alignment π to map AMR edges back to the sentence x , and test whether there exist pairs of crossing edges. Figure 3 shows this mapping for the AMR of Figure 2, where the edge connecting *excuse* to *I* crosses another edge. More formally, consider an AMR edge $e = (u, \ell, v)$. Let $\pi(u) = i$ and $\pi(v) = j$, so that u is aligned with x_i and v is aligned with x_j . The spanning set for e , written $\mathcal{S}(e)$, is the set of all nodes w such that $\pi(w) = k$ and $i < k < j$ if $i < j$ or $j < k < i$ if $j < i$. We say that e is **projective** if, for every node $w \in \mathcal{S}(e)$, all of its parent and child nodes are in $\mathcal{S}(e) \cup \{u, v\}$; otherwise, we say that e is **non-projective**. An AMR is projective if all of its edges are projective, and is non-projective otherwise. This corresponds to the intuitive definition of projectivity for DAGs introduced in Sagae and Tsujii (2008) and is closely related to the definition of non-crossing graphs of Kuhlmann and Jonsson (2015).

Non-projective edges	6%
Non-projective AMRs	51%
Reentrant edges	41%
AMRs with at least one reentrancy	93%

Table 1: Statistics for non-projectivity and reentrancies in 200 AMR manually aligned with the associated sentences.⁵

Table 1 demonstrates that a relatively small percentage of all AMR edges are non-projective. Yet, a large fraction of the sentences contain at least one non-projective edge. Our parser is able to construct non-projective edges, as described in §3.

Reentrancy AMRs are graphs rather than trees because they can have nodes with multiple parents, called reentrant nodes, as in the node *you* for the AMR of Figure 2. There are two phenomena that cause reentrancies in AMR: control, where a reentrant edge appears between siblings of a control verb, and co-reference, where multiple mentions correspond to the same concept.⁶

In contrast, dependency trees do not have nodes with multiple parents. Therefore, when creating a new arc, transition systems for dependency parsing check that the dependent does not already have a head node, preventing the node from having additional parents. To handle reentrancy, which is not uncommon in AMR structures as shown in Table 1, we drop this constraint.

Alignment Another main difference with dependency parsing is that in AMR there is no straightforward mapping between a word in the sentence and a node in the graph: words may generate no nodes, one node or multiple nodes. In addition, the labels at the nodes are often not easily determined by the word in the sentence. For instance *expectation* translates to *expect-01* and *teacher* translates to the two nodes *teach-01* and *person*, connected through an *:ARGO* edge, expressing that a teacher is a person who teaches. A mechanism of concept identification is therefore required to map each token x_i to a subgraph with the correct labels at its nodes and edges: if π is the

⁵https://github.com/jflanigan/jamr/blob/master/docs/Hand_Alignments.md

⁶A valid criticism of AMR is that these two reentrancies are of a completely different type, and should not be collapsed together. Co-reference is a discourse feature, working by extra-semantic mechanisms and able to cross sentence boundaries, which are not crossed in AMR annotation.

gold alignment, this should be the subgraph $\overleftarrow{\pi}(i)$ defined in Equation (1). To obtain alignments between the tokens in the sentence and the nodes in the AMR graph of our training data, we run the JAMR aligner.⁷

3 Transition system for AMR Parsing

A **stack** $\sigma = \sigma_n | \dots | \sigma_1 | \sigma_0$ is a list of nodes of the partially constructed AMR graph, with the top element σ_0 at the right. We use the symbol ‘|’ as the concatenation operator. A **buffer** $\beta = \beta_0 | \beta_1 | \dots | \beta_n$ is a list of indices from x , with the first element β_0 at the left, representing the word tokens from the input still to be processed. A **configuration** of our parser is a triple (σ, β, A) , where A is the set of AMR edges that have been constructed up to this point.

In order to introduce the transition actions of our parser we need some additional notation. We use a function a that maps indices from x to AMR graph fragments. For each $i \in [n]$, $a(i)$ is a graph $G_a = (V_a, E_a)$, with single root $\text{root}(G_a)$, representing the semantic contribution of word x_i to the AMR for x . As already mentioned, G_a can have a single node representing the concept associated with x_i , or it can have several nodes in case x_i denotes a complex concept, or it can be empty.

The transition **Shift** is used to decide if and what to push on the stack after consuming a token from the buffer. Intuitively, the graph fragment $a(\beta_0)$ obtained from the token β_0 , if not empty, is “merged” with the graph we have constructed so far. We then push onto the stack the node $\text{root}(a(\beta_0))$ for further processing. **LARC**(ℓ) creates an edge with label ℓ between the top-most node and the second top-most node in the stack, and pops the latter. **RArc**(ℓ) is the symmetric operation, but does not pop any node from the stack.

Finally, **Reduce** pops the top-most node from the stack, and it also recovers reentrant edges between its sibling nodes, capturing for instance several control verb patterns. To accomplish this, **Reduce** decides whether to create an additional edge between the node being removed and the previously created sibling in the partial graph. With this operation the transition system is able to capture non-projective patterns,⁸ according to the definition given in §2.1, when formed by arcs be-

⁷<https://github.com/jflanigan/jamr>

⁸In an earlier version of this paper this mechanism was not used, yielding a strictly projective parser.

tween nodes that share the same parent. This way of handling control verbs is similar to the *REEN-TRANCE* transition of Wang et al. (2015a).

The choice of popping the dependent in the *LARC* transition is inspired by *ARCEAGER*, where left-arcs are constructed bottom-up to increase the incrementality of the transition system (Nivre, 2004). This affects our ability to recover some reentrant edges: consider a node u with two parents v and v' , where the arc $v \rightarrow u$ is a left-arc and $v' \rightarrow u$ is any arc. If the first arc to be processed is $v \rightarrow u$, we use *LARC* that pops u , hence making it impossible to create the second arc $v' \rightarrow u$. Nevertheless, we discovered that this approach works better than a completely unrestricted allowance of reentrancy. The reason is that if we do not remove dependents at all when first attached to a node, the stack becomes larger, and nodes which should be connected end up being distant from each other, and as such, are never connected.

The initial configuration of the system has a \circ node (representing the root) in the stack and the entire sentence in the buffer. The terminal configuration consists of an empty buffer and a stack with only the \circ node. The transitions required to parse the sentence *The boy and the girl* are shown in Table 2, where the first line shows the initial configuration and the last line shows the terminal configuration.

Similarly to the transitions of the *ARCEAGER*, the above transitions construct edges as soon as the head and the dependent are available in the stack, with the aim of maximizing the parser incrementality. We now show that our greedy transition-based AMR parser is linear-time in n , the length of the input sentence x . We first claim that the output graph has size $\mathcal{O}(n)$. Each token in x is mapped to a constant number of nodes in the graph by *Shift*. Thus the number of nodes is $\mathcal{O}(n)$. Furthermore, each node can have at most three parent nodes, created by transitions *RArc*, *LARC* and *Reduce*, respectively. Thus the number of edges is also $\mathcal{O}(n)$. It is possible to bound the maximum number of transitions required to parse x : the number of *Shift* is bounded by n , and the number of *Reduce*, *LARC* and *RArc* is bounded by the size of the graph, which is $\mathcal{O}(n)$. Since each transition can be carried out in constant time, we conclude that our parser runs in linear time.

4 Training the System

Several components have to be learned: (1) a transition classifier that predicts the next transition given the current configuration, (2) a binary classifier that decides whether or not to create a reentrancy after a *Reduce*, (3) a concept identification step for each *Shift* to compute $a(\beta_0)$, and (3) another classifier to label edges after each *LARC* or *RArc*.

4.1 Oracle

Training our system from data requires an oracle—an algorithm that given a gold-standard AMR graph and a sentence returns transition sequences that maximize the overlap between the gold-standard graph and the graph dictated by the sequence of transitions.

We adopt a shortest stack, static oracle similar to Chen and Manning (2014). Informally, static means that if the actual configuration of the parser has no mistakes, the oracle provides a transition that does not introduce any mistake. Shortest stack means that the oracle prefers transitions where the number of items in the stack is minimized. Given the current configuration (σ, β, A) and the gold-standard graph $G = (V_g, A_g)$, the oracle is defined as follows, where we test the conditions in the given order and apply the action associated with the first match:

1. if $\exists \ell[(\sigma_0, \ell, \sigma_1) \in A_g]$ then *LARC*(ℓ);
2. if $\exists \ell[(\sigma_1, \ell, \sigma_0) \in A_g]$ then *RArc*(ℓ);
3. if $\neg \exists i, \ell[(\sigma_0, \ell, \beta_i) \in A_g \vee (\beta_i, \ell, \sigma_0) \in A_g]$ then *Reduce*;
4. *Shift* otherwise.

The oracle first checks whether some gold-standard edge can be constructed from the two elements at the top of the stack (conditions 1 and 2). If *LARC* or *RArc* are not possible, the oracle checks whether all possible edges in the gold graph involving σ_0 have already been processed, in which case it chooses *Reduce* (conditions 3). To this end, it suffices to check the buffer, since *LARC* and *RArc* have already been excluded and elements in the stack deeper than position two can no longer be accessed by the parser. If *Reduce* is not possible, *Shift* is chosen.

Besides deciding on the next transition, the oracle also needs the alignments, which we generate

action	stack	buffer	edges
-	[o]	[the,boy,and,the,girl]	{}
Shift	[o]	[boy,and,the,girl]	{}
Shift	[o, boy]	[and,the,girl]	{}
Shift	[o, boy, and]	[the,girl]	{}
LArc	[o, and]	[the,girl]	$\{\langle and, :op1, boy \rangle\} = A_1$
RArc	[o, and]	[the,girl]	$A_1 \cup \{\langle o, :top, and \rangle\} = A_2$
Shift	[o, and]	[girl]	A_2
Shift	[o, and, girl]	[]	A_2
RArc	[o, and, girl]	[]	$A_2 \cup \{\langle and, :op2, girl \rangle\} = A_3$
Reduce	[o, and]	[]	A_3
Reduce	[o]	[]	A_3

Table 2: Parsing steps for the sentence “*The boy and the girl.*”

with JAMR, in order to know how to map the next token in the sentence to its AMR subgraph $\overleftarrow{\pi}(i)$ defined in (1).

4.2 Transition Classifier

Like all other transition systems of this kind, our transition system has a “controller” that predicts a transition given the current configuration (among Shift, LArc, RArc and Reduce). The examples from which we learn this controller are based on features extracted from the oracle transition sequences, where the oracle is applied on the training data.

As a classifier, we use a feed-forward neural network with two hidden layers of 200 tanh units and learning rate set to 0.1, with linear decaying. The input to the network consists of the concatenation of embeddings for words, POS tags and Stanford parser dependencies, one-hot vectors for named entities and additional sparse features, extracted from the current configuration of the transition system; this is reported in more details in Table 3. The embeddings for words and POS tags were pre-trained on a large unannotated corpus consisting of the first 1 billion characters from Wikipedia.⁹ For lexical information, we also extract the leftmost (in the order of the aligned words) child (c), leftmost parent (p) and leftmost grandchild (cc). Leftmost and rightmost items are common features for transition-based parsers (Zhang and Nivre, 2011; Chen and Manning, 2014) but we found only leftmost to be helpful in our case. All POS tags, dependencies and named entities are generated using Stanford CoreNLP (Manning et al., 2014). The accuracy of

this classifier on the development set is 84%.

Similarly, we train a binary classifier for deciding whether or not to create a reentrant edge after a Reduce: in this case we use word and POS embeddings for the two nodes being connected and their parent as well as dependency label embeddings for the arcs between them.

4.3 Concept Identification

This routine is called every time the transition classifier decides to do a Shift; it is denoted by $a(\cdot)$ in §3. This component could be learned in a supervised manner, but we were not able to improve on a simple heuristic, which works as follows: during training, for each Shift decided by the oracle, we store the pair $(\beta_0, \overleftarrow{\pi}(i))$ in a phrase-table. During parsing, the most frequent graph H for the given token is then chosen. In other words, $a(i)$ approximates $\overleftarrow{\pi}(i)$ by means of the graph most frequently seen among all occurrences of token x_i in the training set.

An obvious problem with the phrase-table approach is that it does not generalize to unseen words. In addition, our heuristic relies on the fact that the mappings observed in the data are correct, which is not the case when the JAMR-generated alignments contain a mistake. In order to alleviate this problem we observe that there are classes of words such as named entities and numeric quantities that can be disambiguated in a deterministic manner. We therefore implement a set of “hooks” that are triggered by the named entity tag of the next token in the sentence. These hooks override the normal Shift mechanism and apply a fixed rule instead. For instance, when we see the token *New York* (the two tokens are collapsed in a single one

⁹<http://mattmahoney.net/dc/enwik9.zip>

depth	$d(\sigma_0), d(\sigma_1)$
children	$\#c(\sigma_0), \#c(\sigma_1)$
parents	$\#p(\sigma_0), \#p(\sigma_1)$
lexical	$w(\sigma_0), w(\sigma_1), w(\beta_0), w(\beta_1),$ $w(p(\sigma_0)), w(c(\sigma_0)), w(cc(\sigma_0)),$ $w(p(\sigma_1)), w(c(\sigma_1)), w(cc(\sigma_1))$
POS	$s(\sigma_0), s(\sigma_1), s(\beta_0), s(\beta_1)$
entities	$e(\sigma_0), e(\sigma_1), e(\beta_0), e(\beta_1)$
dependency	$\ell(\sigma_0, \sigma_1), \ell(\sigma_1, \sigma_0),$ $\forall i \in \{0, 1\}: \ell(\sigma_i, \beta_0), \ell(\beta_0, \sigma_i)$ $\forall i \in \{1, 2, 3\}: \ell(\beta_0, \beta_i), \ell(\beta_i, \beta_0)$ $\forall i \in \{1, 2, 3\}: \ell(\sigma_0, \beta_i), \ell(\beta_i, \sigma_0)$

Table 3: Features used in transition classifier. The function d maps a stack element to the depth of the associated graph fragment. The functions $\#c$ and $\#p$ count the number of children and parents, respectively, of a stack element. The function w maps a stack/buffer element to the word embedding for the associated word in the sentence. The function p gives the leftmost (according to the alignment) parent of a stack element, the function c the leftmost child and the function cc the leftmost grandchild. The function s maps a stack/buffer element to the part-of-speech embedding for the associated word. The function e maps a stack/buffer element to its entity. Finally, the function ℓ maps a pair of symbols to the dependency label embedding, according to the edge (or lack of) in the dependency tree for the two words these symbols are mapped to.

at preprocessing) we generate the subgraph of Figure 4 and push its root onto the stack. Similar subgraphs are generated for all states, cities, countries and people. We also use hooks for ordinal numbers, percentages, money and dates.

4.4 Edge Labeling

Edge labeling determines the labels for the edges being created. Every time the transition classifier decides to take an LARC or RARC operation, the edge labeler needs to decide on a label for it. There are more than 100 possible labels such as $:ARG0$, $:ARG0-of$, $:ARG1$, $:location$, $:time$ and $:polarity$. We use a feed-forward neural network similar to the one we trained for the transition classifier, with features shown in Table 4. The accuracy of this classifier on the development set is 77%.

Labeling Rules Sometimes the label predicted by the neural network is not a label that satisfies

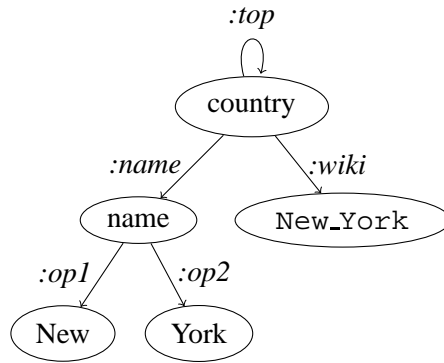


Figure 4: Subgraph for “New York.”

name	feature template
depth	$d(\sigma_0), d(\sigma_1)$
children	$\#c(\sigma_0), \#c(\sigma_1)$
parents	$\#p(\sigma_0), \#p(\sigma_1)$
lexical	$w(\sigma_0), w(\sigma_1),$ $w(p(\sigma_0)), w(c(\sigma_0)), w(cc(\sigma_0)),$ $w(p(\sigma_1)), w(c(\sigma_1)), w(cc(\sigma_1))$
POS	$s(\sigma_0), s(\sigma_1)$
entities	$e(\sigma_0), e(\sigma_1)$
dependency	$\ell(\sigma_0, \beta_0), \ell(\beta_0, \sigma_0)$

Table 4: Features used in edge labeling. See Table 3 for a legend of symbols.

the requirements of AMR. For instance, the label $:top$ can only be applied when the node from which the edge starts is the special \circ node. In order to avoid generating such erroneous labels, we use a set of rules, shown in Table 5. These rules determine which labels are allowed for the newly created edge so that we only consider those during prediction. Also ARG roles cannot always be applied: each Propbank frame allows a limited number of arguments. For example, while $add-01$ and $add-02$ allow for $:ARG1$ and $:ARG2$ (and their inverse $:ARG1-of$ and $:ARG2-of$), $add-03$ and $add-04$ only allow $:ARG2$ (and $:ARG2-of$).

5 Fine-grained Evaluation

Until now, AMR parsers were evaluated using the Smatch score.¹⁰ Given the candidate graphs and the gold graphs in the form of AMR annotations, Smatch first tries to find the best alignments between the variable names for each pair of graphs and it then computes precision, recall and F1 of the

¹⁰Since Smatch is an approximate randomized algorithm, decimal points in the results vary between different runs and are not reported. This approach was also taken by Wang et al. (2015b) and others.

label	ex.	start	end
:top	Yes	o	
:polarity	Yes		-
:mode	Yes		inter. expr. imp.
:value	No		"\w+" [0-9] ⁺
:day	No	d-ent	[1 2 \dots 31] ⁺
:month	No	d-ent	[1 2 \dots 12] ⁺
:year	No	d-ent	[0-9] ⁺
:decade	No	d-ent	[0-9] ⁺
:century	No	d-ent	[0-9] ⁺
:weekday	Yes	d-ent	[monday \dots sunday]
:quarter	No	d-ent	[1 2 3 4] ⁺
:season	Yes	d-ent	[winter fall spring summer] ⁺
:timezone	Yes	d-ent	[A-Z] ³

Table 5: Labeling rules: For each edge label, we provide regular expressions that must hold on the labels at the start node (start) and the end node (end) of the edge. Ex. indicates when the rule is exclusive, d-ent is the AMR concept *date-entity*, inter. is the AMR constant *interrogative*, expr. is the AMR constant *expressive*, imp. is the AMR constant *imperative*.

concepts and relations. We note that the Smatch score has two flaws: (1) while AMR parsing involves a large number of subtasks, the Smatch score consists of a single number that does not assess the quality of each subtasks separately; (2) the Smatch score weighs different types of errors in a way which is not necessarily useful for solving a specific NLP problem. For example, for a specific problem concept detection might be deemed more important than edge detection, or guessing the wrong sense for a concept might be considered less severe than guessing the wrong verb altogether.

Consider the two parses for the sentence *Silvio Berlusconi gave Lucio Stanca his current role of modernizing Italy's bureaucracy* in Figure 5. At the top, we show the output of a parser (*Parse 1*) that is not able to deal with named entities. At the bottom, we show the output of a parser (*Parse 2*) which, except for *:name*, *:op* and *:wiki*, always uses the edge label *:ARG0*. The Smatch scores for the two parses are 56 and 78 respectively. Both parses make obvious mistakes but the three named entity errors in *Parse 1* are considered more important than the six wrong labels in *Parse 2*. However, without further analysis, it is not advisable to conclude that *Parse 2* is better than *Parse 1*. In order to better understand the limitations of the differ-

```
(g / give-01
  :ARG0 (p3 / silvio :mod (n4 / berlusconi))
  :ARG1 (r / role
    :time (c2 / current)
    :mod (m / modernize-01
      :ARG0 p4
      :ARG1 (b / bureaucracy :part-of (c3 / italy)))
    :poss p4)
  :ARG2 (p4 / person lucio :mod stanca))
```

```
(g / give-01
  :ARG0 (p3 / person :wiki "Silvio Berlusconi"
    :name (n4 / name :op1 "Silvio" :op2 "Berlusconi"))
  :ARG0 (r / role
    :ARG0 (c2 / current)
    :ARG0 (m / modernize-01
      :ARG0 p4
      :ARG0 (b / bureaucracy
        :ARG0 (c3 / country :wiki "Italy"
          :name (n6 / name :op1 "Italy"))))
    :ARG0 p4)
  :ARG0 (p4 / person :wiki -
    :name (n5 / name :op1 "Lucio" :op2 "Stanca")))
```

Figure 5: Two parses for the sentence “*Silvio Berlusconi gave Lucio Stanca his current role of modernizing Italy’s bureaucracy.*”

ent parsers, find their strengths and gain insight in which downstream tasks they may be helpful, we compute a set of metrics on the test set.

Unlabeled is the Smatch score computed on the predicted graphs after removing all edge labels. In this way, we only assess the node labels and the graph topology, which may be enough to benefit several NLP tasks because it identifies basic predicate-argument structure. For instance, we may be interested in knowing whether two events or entities are related to each other, while not being concerned with the precise type of relation holding between them.

No WSD gives a score that does not take into account word sense disambiguation errors. By ignoring the sense specified by the Propbank frame used (e.g., *duck-01* vs *duck-02*) we have a score that does not take into account this additional complexity in the parsing procedure. To compute this score, we simply strip off the suffixes from all Propbank frames and calculate the Smatch score.

Following Sawai et al. (2015), we also evaluate the parsers using the Smatch score on noun phrases only (**NP-only**), by extracting from the AMR dataset all noun phrases that do not include further NPs.

As we previously discussed, reentrancy is a very important characteristic of AMR graphs and it is not trivial to handle. We therefore implement a test for it (**Reentrancy**), where we compute the Smatch score only on reentrant edges.

Concept identification is another critical component of the parsing process and we therefore

compute the F-score on the list of predicted concepts (**Concepts**) too. Identifying the correct concepts is fundamental: if a concept is not identified, it will not be possible to retrieve any edge involving that concept, with likely significant consequences on accuracy. This metric is therefore quite important to score highly on.

Similarly to our score for concepts, we further compute an F-score on the named entities (**Named Ent.**) and wiki roles for named entities (**Wikification**) that consider edges labeled with *:name* and *:wiki* respectively. These two metrics are strictly related to the concept score. However, since named entity recognition is the focus of dedicated research, we believe it is important to define a metric that specifically assesses this problem. Negation detection is another task which has received some attention. An F-score for this (**Negations**) is also defined, where we find all negated concepts by looking for the *:polarity* role. The reason we can compute a simple F-score instead of using Smatch for these metrics is that there are no variable names involved.

Finally we compute the Smatch score on *:ARG* edges only, in order to have a score for semantic role labeling (**SRL**), which is another extremely important subtask of AMR, as it is based on the identification of predicate-argument structures.

Using this evaluation suite we can evaluate AMRs on a wide range of metrics that can help us find strengths and weakness of each parser, hence speeding up the research in this area. Table 6 reports the scores for the two parses of Figure 5, where we see that *Parse 1* gets a high score for semantic role labeling while *Parse 2* is optimal for named entity recognition. Moreover, we can make additional observations such as that *Parse 2* is optimal with respect to unlabeled score and that *Parse 1* recovers more reentrancies.

6 Experiments

We compare our parser¹¹ against two available parsers: JAMR (Flanigan et al., 2014) and CAMR (Wang et al., 2015b; Wang et al., 2015a), using the LDC2015E86 dataset for evaluation. Both parsers

¹¹Our parser is available at <https://github.com/mdtux89/amr-eager>, the evaluation suite at <https://github.com/mdtux89/amr-evaluation> and a demo at <http://cohort.inf.ed.ac.uk/amreager.html>

Metric	First parse	Second parse
Smatch	56	78
Unlabeled	65	100
No WSD	56	78
NP-only	39	86
Reentrancy	69	46
Concepts	56	100
Named Ent.	0	100
Wikification	0	100
Negations	0	0
SRL	69	54

Table 6: Evaluation of the two parses in Figure 5 with the proposed evaluation suite.

Metric	J'14	C'15	J'16	Ours
Smatch	58	63	67	64
Unlabeled	61	69	69	69
No WSD	58	64	68	65
NP-only	47	54	58	55
Reentrancy	38	41	42	41
Concepts	79	80	83	83
Named Ent.	75	75	79	83
Wikification	0	0	75	64
Negations	16	18	45	48
SRL	55	60	60	56

Table 7: Results on test split of LDC2015E86 for JAMR, CAMR and our AMREAGER. J stands for JAMR and C for CAMR (followed by the year of publication). Best systems are in bold.

are available online¹² and were recently updated for SemEval-2016 Task 8 (Flanigan et al., 2016; Wang et al., 2016). However, CAMR’s SemEval system, which reports a Smatch score of 67, is not publicly available. CAMR has a quadratic worst-case complexity (although linear in practice). In JAMR, the concept identification step is quadratic and the relation identification step is $O(|V|^2 \log |V|)$, with $|V|$ being the set of nodes in the AMR graph.

Table 7 shows the results obtained by the parsers on all metrics previously introduced. On Smatch, our system does not give state-of-the-art results. However, we do obtain the best results for *Unlabeled* and *Concept* and outperform the other parses for *Named Ent.* and *Negations*. Our score of *Reentrancy* is also close the best scoring system, which is particularly relevant given

¹²JAMR: <https://github.com/jflanigan/jamr>, CAMR: <https://github.com/c-amr/camr>.

the importance of reentrancies in AMR. The use of the **Reduce** transition, which targets reentrancies caused by control verbs, is critical in order to achieve this result.

The relatively high results we obtain for the unlabeled case suggests that our parser has difficulty in labeling the arcs. Our score for concept identification, which is on par with the best result from the other parsers, demonstrates that there is a relatively low level of token ambiguity. State-of-the-art results for this problem can be obtained by choosing the most frequent subgraph for a given token based on a phrase-table constructed from JAMR alignments on the training data. The scores for named entities and wikification are heavily dependent on the hooks mentioned in §4.3, which in turn relies on the named entity recognizer to make the correct predictions. In order to alleviate the problem of wrong automatic alignments with respect to polarity and better detect negation, we performed a post-processing step on the aligner output where we align the AMR constant - (minus) with words bearing negative polarity such as *not*, *illegitimate* and *asymmetry*.

Our experiments demonstrate that there is no parser for AMR yet that conclusively does better than all other parsers on all metrics. Advantages of our parser are the worst-case linear complexity and the fact that it is possible to perform incremental AMR parsing, which is both helpful for real-time applications and to investigate how meaning of English sentences can be built incrementally left-to-right.

7 Related Work

The first data-driven AMR parser is due to Flanigan et al. (2014). The problem is addressed in two separate stages: concept identification and relation identification. They use a sequence labeling algorithm to identify concepts and frame the relation prediction task as a constrained combinatorial optimization problem. Werling et al. (2015) notice that the difficult bit is the concept identification and propose a better way to handle that task: an action classifier to generate concepts by applying predetermined actions. Other proposals involve a synchronous hyperedge replacement grammar solution (Peng et al., 2015), a syntax-based machine translation approach (Pust et al., 2015) where a grammar of string-to-tree rules is created after reducing AMR graphs to trees by re-

moving all reentrancies, a CCG system that first parses sentences into lambda-calculus representations (Artzi et al., 2015). A systematic translation from AMR to first order logic formulas, with a special treatment for quantification, reentrancy and negation, is discussed in Bos (2016). In Vanderwende et al. (2015), a pre-existing logical form parser is used and the output is then converted into AMR graphs. Yet another solution is proposed by Rao et al. (2015) who discuss a parser that uses SEARN (Daumé III et al., 2009), a “learning to search” algorithm.

Transition-based algorithms for AMR parsing are compelling because traditional graph-based techniques are computationally expensive. Wang et al. (2015b) and Wang et al. (2015a) propose a framework that parses a sentence into its AMR structure through a two-stage process: a dependency tree is generated from the input sentence through a transition-based parser and then another transition-based parser is used to generate the AMR. The main benefit of this approach is that the dependency parser can be trained on a training set much larger than the training set for the tree-to-graph algorithm. Others further built on this parser: Goodman et al. (2016) use imitation learning to alleviate the problem of error propagation in the greedy parser, while Barzdins and Gosko (2016) create a wrapper around it to fix frequent mistakes and investigate ensembles with a character level neural parser. More recently Zhou et al. (2016) presented a non-greedy transition system for AMR parsing, based on ARCSTANDARD (Nivre, 2004).

AMR parsing as a whole is a complex task because it involves many subtasks including named entity recognition, co-reference resolution and semantic role labeling. Sawai et al. (2015) do not attempt at parsing AMR graphs for entire sentences but they instead handle simple noun phrases (NPs). They extract NPs from the AMR dataset only when they do not include further NPs, do not include pronouns nor named entities. Due to these restrictions, the AMRs are mostly trees and easier to handle than the original AMR graphs. They approach this task using a transition based system inspired by ARCSTANDARD.

AMR is not the only way to represent meaning in natural language sentences. Alternative semantic representations have been developed and studied, such as Boxer (Bos et al., 2004), CCG (Steed-

man, 1996; Steedman, 2000) and UCCA (Abend and Rappoport, 2013).

8 Conclusion

We presented a transition system that builds AMR graphs in linear time by processing the sentences left-to-right, trained with feed-forward neural networks. The parser demonstrates that it is possible to perform AMR parsing using techniques inspired by dependency parsing.

We also noted that it is less informative to evaluate the entire parsing process with Smatch than to use a collection of metrics aimed at evaluating the various subproblems in the parsing process. We further showed that our left-to-right transition system is competitive with publicly available state-of-the-art parsers. Although we do not outperform the best baseline in terms of Smatch score, we show on par or better results for several of the metrics proposed. We hope that moving away from a single-metric evaluation will further speed up progress in AMR parsing.

Acknowledgments

The authors would like to thank the three anonymous reviewers and Sameer Bansal, Jeff Flanigan, Sorcha Gilroy, Adam Lopez, Nikos Papasaran-topoulos, Nathan Schneider, Mark Steedman, Sam Thomson, Clara Vania and Chuan Wang for their help and comments. This research was supported by a grant from Bloomberg and by the H2020 project SUMMA, under grant agreement 688139.

References

- Omri Abend and Ari Rappoport. 2013. Universal conceptual cognitive annotation (UCCA). In *Proceedings of ACL*.
- Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. *Proceedings of EMNLP*.
- 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. *Proceedings of Linguistic Annotation Workshop*.
- 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. *arXiv preprint arXiv:1604.01278*.
- Johan Bos, Stephen Clark, Mark Steedman, James R Curran, and Julia Hockenmaier. 2004. Wide-coverage semantic representations from a ccg parser. In *Proceedings of COLING*. Association for Computational Linguistics.
- Johan Bos. 2016. Expressive power of abstract meaning representations. *Computational Linguistics*, 42.
- Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. *Proceedings of ACL*.
- Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of EMNLP*.
- Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine learning*, 75(3):297–325.
- Yantao Du, Fan Zhang, Weiwei Sun, and Xiaojun Wan. 2014. Peking: Profiling syntactic tree parsing techniques for semantic graph parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval-2014)*, pages 459–464.
- Jeffrey Flanigan, Sam Thomson, Jaime G Carbonell, Chris Dyer, and Noah A Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. *Proceedings of ACL*.
- 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.
- James Goodman, Andreas Vlachos, and Jason Naradowsky. 2016. Noise reduction and targeted exploration in imitation learning for abstract meaning representation parsing. *Proceedings of ACL*.
- Marco Kuhlmann and Peter Jonsson. 2015. Parsing to noncrossing dependency graphs. *Transactions of the Association for Computational Linguistics*, pages 559–570.
- Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *ACL System Demonstrations*.
- Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*. ACL.
- Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics, Volume 34, Number 4, December 2008*.
- Xiaochang Peng, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. *Proceedings of CoNLL*.

- Michael Pust, Ulf Hermjakob, Kevin Knight, Daniel Marcu, and Jonathan May. 2015. Using syntax-based machine translation to parse english into abstract meaning representation. *arXiv preprint arXiv:1504.06665*.
- Sudh Rao, Yogarshi Vyas, Hal Daume III, and Philip Resnik. 2015. Parser for abstract meaning representation using learning to search. *arXiv:1510.07586*.
- Corentin Ribeyre, Éric Villemonte de La Clergerie, and Djamé Seddah. 2015. Because syntax does matter: Improving predicate-argument structures parsing using syntactic features. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.
- Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-reduce dependency DAG parsing. *Proceedings of COLING*.
- Yuichiro Sawai, Hiroyuki Shindo, and Yuji Matsumoto. 2015. Semantic structure analysis of noun phrases using abstract meaning representation. *Proceedings of ACL*.
- Mark Steedman. 1996. *Surface Structure and Interpretation*. The MIT Press.
- Mark Steedman. 2000. *The Syntactic Process*. The MIT Press.
- Lucy Vanderwende, Arul Menezes, and Chris Quirk. 2015. An AMR parser for english, french, german, spanish and japanese and a new AMR-annotated corpus. *Proceedings of NAACL-HLT*.
- Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015a. Boosting transition-based AMR parsing with refined actions and auxiliary analyzers. *Proceedings of ACL*.
- Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015b. A transition-based algorithm for AMR parsing. *Proceedings of NAACL*.
- Chuan Wang, Sameer Pradhan, Nianwen Xue, Xiaoman Pan, and Heng Ji. 2016. CAMR at SemEval-2016 task 8: An extended transition-based AMR parser. *Proceedings of SemEval*.
- Keenon Werling, Gabor Angeli, and Christopher Manning. 2015. Robust subgraph generation improves abstract meaning representation parsing. *arXiv preprint arXiv:1506.03139*.
- Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. *Proceedings of ACL*.
- 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*.