DeepCx: A transition-based approach for shallow semantic parsing with complex constructional triggers

Jesse Dunietz
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jdunietz@cs.cmu.edu

Jaime Carbonell and **Lori Levin**
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{jgc,lsl}@cs.cmu.edu

Abstract

This paper introduces the SURFACE CONSTRUCTION LABELING (SCL) task, which expands the coverage of Shallow Semantic Parsing (SSP) to include frames triggered by complex constructions. We present DeepCx, a neural, transition-based system for SCL. As a test case for the approach, we apply DeepCx to the task of tagging causal language in English, which relies on a wider variety of constructions than are typically addressed in SSP. We report substantial improvements over previous tagging efforts on a causal language dataset. We also propose ways DeepCx could be extended to still more difficult constructions and to other semantic domains once appropriate datasets become available.

1 Introduction

Shallow semantic parsing (SSP) aims to tag the triggers of semantic relations and the phrases between which those relations hold. However, words are not the only bearers of relational meaning: multi-word expressions (MWEs) and even arbitrarily complex constructions can express relations and evoke semantic frames (see, e.g., Fillmore et al., 2012). For example, causation, concession, and comparison are frequently expressed using complex constructions (see Table 1). MWE research has made strides in identifying MWE strings (see, e.g., Baldwin and Kim, 2010), but little work has addressed tagging arguments of such constructional triggers; many of the examples in Table 1 remain a challenge for conventional SSP.

This paper introduces the broader task of SURFACE CONSTRUCTION LABELING (SCL; §3). Like SSP, SCL aims to tag the surface elements of a sentence that express and participate in relational meanings. But in SCL, triggers are not just words or lexical units, but instances of constructions of the sort described by CONSTRUCTION GRAMMAR (CxG; Fillmore et al., 1988; Goldberg, 1995): arbitrarily complex sets of tokens that carry meaning. These constructions can consist of single words, fixed MWEs, gappy MWEs, or even grammatical patterns.

(1) WE must `regulate` to **inhibit** *unsound practices*.
(2) THIS **opens the way to** *applying the law more widely*.
(3) Judy's comments were **SO** OFFENSIVE that *I left*.

(4) We headed out **in spite of** the bad weather.
(5) We value any contribution, **no matter** its size.
(6) Strange **as** it seems, there's been a run of crazy dreams!

(7) **More** boys wanted to play **than** girls.
(8) Andrew is **as** annoying **as** he **is** useless.
(9) I'm poor**er than** I'd like.

Table 1: Examples of causal (1–3), concessive (4–6), and comparative (7–9) constructions, with triggers bolded. Arguments of causal examples are annotated as in the BECAUSE annotation scheme, with CAUSES in blue small caps, *effects* in red italics, and `means` in purple typewriter text.

We propose a transition system for SCL that can tag multi-word, possibly gappy sequences of tokens as triggers and arguments (§4). As a test case for the approach, we address English causal language, a valuable target for semantic analysis in its own right. (Extensions to arbitrary frame and role types would be straightforward with appropriate data; see §8.) The transitions can handle most types of constructions, including those with multiple arguments, missing arguments, and even triggers that overlap or are interleaved with arguments. We present DeepCx, a neural network that tags causal language using this transition system (§5). We also describe experiments applying DeepCx to the BECAUSE corpus (§6), showing that DeepCx significantly outperforms prior construction-based work on predicting causal frames (§7). Finally, we discuss how the transition system and tagger model could be adapted to more difficult SCL tasks (§8).

2 Background and related work

2.1 Shallow semantic parsing

SCL of course inherits from SSP, which has a venerable tagging tradition. For PropBank data, dozens of taggers have been developed (see Carreras and Màrquez, 2004, 2005; Surdeanu et al., 2008; Hajič et al., 2009). These typically focus on argument tagging, since PropBank triggers are readily identified by their POS tags. One popular design is a multi-stage pipeline that identifies argument spans and then labels them. Another alternative is BIO-style classification of argument words, either with conventional classifiers or with neural networks (e.g., Collobert et al., 2011; Foland and Martin, 2015). More recent systems (e.g., Täckström et al., 2015; Roth and Lapata, 2016) use neural networks to score and label possible argument spans or heads.

FrameNet-based tagging is more difficult, as triggers must be identified and disambiguated. Many FrameNet taggers have taken a pipeline approach (see, e.g., Baker et al., 2007; Das et al., 2014) in which targets are first identified with a whitelist or simple rules. They are then assigned frames, which determine the available frame elements, and finally the frame elements are identified and labeled. Again, neural networks have also been used to score argument spans and heads (Täckström et al., 2015; FitzGerald et al., 2015; Roth, 2016).

Systems in both paradigms are constrained by their underlying representations. PropBank covers only verbs and certain nominal and adjectival predicates. FrameNet's frame-evoking elements are broader, including verbs, prepositions, adverbs, conjunctions, and even some MWEs, but must still be single words or MWEs that act like words. As Table 1 demonstrates, some semantic domains, such as causality, demand a more flexible approach.

2.2 Construction grammar

CxG, which posits that the fundamental units of language are CONSTRUCTIONS—pairings of meanings with arbitrary linguistic forms. For instance, *so X that Y* (example 3) is characterized by a single construction, where the form is *so* ⟨*adjective X*⟩ ⟨*finite clausal complement Y*⟩ and the meaning is *X to an extreme that causes Y*. Following Dunietz et al. (2017a,b), we borrow two core insights of CxG: first, that morphemes, words, MWEs, and grammar are **all on equal footing** as "learned pairings of form and function" (Goldberg, 2013); and second, that constructions pair patterns of surface forms **directly with meanings**. Thus, we can tag any surface realizations of constructions as meaning-bearing triggers (hence "surface construction labeling").

2.3 Causal language

To test the SCL approach, we examine causal language, which conveys essential semantic information and is especially rich in constructional triggers.

Our data representation for causal language comes from the BECAUSE 2.1 corpus (Dunietz et al., 2017b), which focuses on what causal meanings are explicitly stated in the text. It defines causal language as any construction which presents one event, state, action, or entity as promoting or hindering another, and which includes at least one lexical trigger to anchor the annotation. For each instance of causal language, up to three spans are annotated: the **causal connective** (the trigger of the causal relation), the **cause** span, the **effect** span, and occasionally a **means** span (if a means by which the cause produces the effect is specified). See Table 1 for examples of analyses of causal language under the BECAUSE scheme. The corpus includes 4,867 sentences (123,674 tokens) of news articles and Congressional hearing transcripts fully annotated for causal language.

The only prior work on construction-based semantic parsing that we know of is Causeway (Dunietz et al., 2017a), also based on the BECAUSE corpus. Causeway detects causal connectives using lexico-syntactic patterns, then applies heuristics and classifiers to tag arguments and remove false positives. It achieves moderate performance, but requires extensive tuning and feature engineering.

2.4 Transition-based systems

Transition-based systems have primarily been used for dependency parsing (e.g., Nivre et al., 2007; Nivre, 2008; Chen et al., 2014; Choi and Palmer, 2011a). Indeed, our system borrows many implementation elements from Dyer et al. (2015), who describe a shift-reduce parser that embeds the stack and buffer as LSTMs. This parser employs the novel STACK LSTM data structure—an LSTM augmented with a stack pointer, enabling it to be rewound to a previous state.

Transition systems have been developed for semantic tasks, as well. Titov et al. (2009), Henderson et al. (2008), and Swayamdipta et al. (2016) explore extensions of dependency parsing that interleave semantic parsing actions with syntactic

parsing actions. Google's SLING (Ringgaard et al., 2017) applies a custom-designed transition scheme for frame-based parsing and coreference resolution. Vilares and Gómez-Rodríguez (2018) develop a transition system for Abstract Meaning Representation parsing, and TUPA (Hershcovich et al., 2017) does the same for Universal Conceptual Cognitive Annotation. Both can handle discontinuous or reentrant graph structures. Most directly relevant to DeepCx is Choi and Palmer's (2011b) work, which defines a novel transition system for PropBank parsing. Our similar scheme for parsing causal constructions builds on this one, extending it for cases where the spans are not contiguous.

3 The SCL task for causal language

An SCL task closely resembles an SSP task, except that the triggers can be complex constructions. As a corollary, the arguments can also be discontinuous and/or overlap with each other or the trigger.

In the case of causal language, we define the task as reproducing the core elements of the BECAUSE scheme: connective, cause, effect, and means spans. Following Dunietz et al. (2017a), we split the task into two parts: discovering causal connectives (**connective discovery**) and delimiting and labeling the arguments (**argument ID**). Producing the additional metadata that BECAUSE records for each instance is left to future work.

Each span is defined as a set of tokens. This excludes sublexical constructions; we return to this limitation in §8.

4 Transition system

Like Choi and Palmer, DeepCx's transition system first searches for a connective word, and once it has found one, compares it with each word to the right and to the left. In each comparison, it selects a transition that labels the word as unrelated to the current connective word, as another connective word (or FRAGMENT), or as a member of some argument span(s). Once all words have been compared to the current connective, the system advances to the next possible initial connective word. In the worst case, then, each sentence takes $O(n^2)$ transitions.

Table 2 gives the full set of transitions. The transitions act on a state tuple $(\lambda_1, \lambda_2, a, \lambda_3, \lambda_4, s, A)$. a is the index of the current possible "connective anchor"—the word being tentatively treated as the initial (i.e., leftmost) word of a connective. λ_1 is the list of word indices to a's left that have not yet been compared with it, and λ_2 represents the words to the left of a that have already been compared. Likewise, λ_3 and λ_4 contain the indices of compared and uncompared words, respectively, to the right. Thus, words move from λ_1 to λ_2 and from λ_4 to λ_3 as they are compared with a. s is a boolean indicating whether we are currently comparing words in the sentence to a, i.e., whether a has been confirmed as a connective anchor. A is a set of partially-constructed causal language instances. Each instance consists of a set of connective word indices plus one set of argument word indices for each argument type. For formal description, we represent A as a set of labeled arcs. The head a of each arc is the connective anchor of a causal language instance i (an arbitrary identifier). The label of the arc indicates what role the tail t plays with respect to i: Cause, Effect, or Means if t is a member of the corresponding argument span, and Frag if t is a connective fragment other than a.

As the algorithm scans from left to right, it assigns a to each word index in turn. If it decides a is not a connective anchor, it issues a NO-CONN and moves on. If a is deemed to start a connective, a new instance is initialized with a NEW-CONN. DeepCx proceeds to compare a with each word to its left, in right-to-left order (i.e., starting from the closest word), then each word to the right (in left-to-right order). For each comparison, it issues a LEFT/RIGHT-ARC, CONN-FRAG, or NO-ARC, depending on whether the comparison word is deemed part of an argument, part of the connective, or neither. For simplicity, we always consider the leftmost connective word to anchor the connective, so all CONN-FRAG transitions occur between a connective word and a word to its right. After all words have been compared with a (i.e., once λ_1 and λ_4 are empty), an automatic SHIFT transition advances a to the next connective anchor candidate.

The initial state is: $\lambda_1 = \lambda_2 = \lambda_3 = [\,]$, $a = 1$, $\lambda_4 = [w_1 \,..\, w_n]$, $s =$ f, and $A = \varnothing$, where w_i is the ith word in the sentence. The algorithm terminates when $a = n$ and either $\lambda_3 = \lambda_4 = [\,]$ or s is false—i.e., when no words remain to a's right, and either a is not a connective anchor or all words in the sentence have been compared with it. An example transition sequence is shown in Table 3.

Some transitions have preconditions, shown inline Table 2 in a smaller font. In addition, several transitions have constraints on their ordering to ensure semantic well-formedness. These constraints

Transition schema	Effect and preconditions
NO-CONN	$(\boldsymbol{\lambda_1}, \lambda_2{=}[], \boldsymbol{a}, \lambda_3{=}[], [\boldsymbol{w} \mid \boldsymbol{\lambda_4}], s{=}\mathsf{f}, A) \Rightarrow ([\boldsymbol{\lambda_1} \mid \boldsymbol{a}], \lambda_2, \boldsymbol{w}, \lambda_3, \boldsymbol{\lambda_4}, s, A)$
NEW-CONN	$(\lambda_1, \lambda_2, a, \lambda_3, \lambda_4, \boldsymbol{s}{=}\mathsf{f}, A) \Rightarrow (\lambda_1, \lambda_2, a, \lambda_3, \lambda_4, \mathbf{t}, A)$
NO-ARC-LEFT	$([\boldsymbol{\lambda_1} \mid \boldsymbol{w}], \boldsymbol{\lambda_2}, a, \lambda_3{=}[], \lambda_4, s{=}\mathsf{t}, A) \Rightarrow (\boldsymbol{\lambda_1}, [\boldsymbol{w} \mid \boldsymbol{\lambda_2}], a, \lambda_3, \lambda_4, s, A)$
NO-ARC-RIGHT	$(\lambda_1{=}[], \lambda_2, a, \boldsymbol{\lambda_3}, [\boldsymbol{w} \mid \boldsymbol{\lambda_4}], s{=}\mathsf{t}, A) \Rightarrow (\lambda_1, \lambda_2, a, [\boldsymbol{\lambda_3} \mid \boldsymbol{w}], \boldsymbol{\lambda_4}, s, A)$
LEFT-ARC$_x$	$([\boldsymbol{\lambda_1} \mid \boldsymbol{w}], \boldsymbol{\lambda_2}, a, \lambda_3{=}[], \lambda_4, s{=}\mathsf{t}, \boldsymbol{A}) \Rightarrow (\boldsymbol{\lambda_1}, [\boldsymbol{w} \mid \boldsymbol{\lambda_2}], a, \lambda_3, \lambda_4, s, \boldsymbol{A} \cup \{\boldsymbol{a} \xrightarrow{\boldsymbol{x_i}} \boldsymbol{w}\})$
RIGHT-ARC$_x$	$(\lambda_1{=}[], \lambda_2, a, \boldsymbol{\lambda_3}, [\boldsymbol{w} \mid \boldsymbol{\lambda_4}], s{=}\mathsf{t}, \boldsymbol{A}) \Rightarrow (\lambda_1, \lambda_2, a, [\boldsymbol{\lambda_3} \mid \boldsymbol{w}], \boldsymbol{\lambda_4}, s, \boldsymbol{A} \cup \{\boldsymbol{a} \xrightarrow{\boldsymbol{x_i}} \boldsymbol{w}\})$
CONN-FRAG	$(\lambda_1{=}[], \lambda_2, a{\neq}\,w, \lambda_3, [w \mid \lambda_4], s{=}\mathsf{t}, \boldsymbol{A}) \Rightarrow (\lambda_1, \lambda_2, a, \lambda_3, [w \mid \lambda_4], s{=}\mathsf{t}, \boldsymbol{A} \cup \{\boldsymbol{a} \xrightarrow{\mathbf{FRAG}_i} \boldsymbol{w}\})$
SHIFT	$(\boldsymbol{\lambda_1}{=}[], \boldsymbol{\lambda_2}, \boldsymbol{a}, [\boldsymbol{w} \mid \boldsymbol{\lambda_3}], \boldsymbol{\lambda_4}{=}[], \boldsymbol{s}{=}\mathsf{t}, A) \Rightarrow (\boldsymbol{\lambda_2}, [\,], \boldsymbol{w}, [\,], \boldsymbol{\lambda_3}, \mathbf{f}, A)$
SPLIT	See text (§4.1)

Table 2: The DeepCx transitions. Pre- and post-transition states are expressed as tuples $(\lambda_1, \lambda_2, a, \lambda_3, \lambda_4, s, A)$. x stands for Cause, Effect, Means, or any combination thereof. i indicates the instance under construction; thus, x_i denotes an argument or fragment arc of instance i. Elements changed by the transition are bolded. Preconditions (small font in the starting states) enforce a consistent transition order by delaying rightward actions until all leftward actions are completed.

Transition	$\boldsymbol{\lambda_1}$	$\boldsymbol{\lambda_2}$	$\boldsymbol{a}$	$\boldsymbol{\lambda_3}$	$\boldsymbol{\lambda_4}$	$\boldsymbol{s}$	$\boldsymbol{A}$
–	[]	[]	1	[]	[1 . . 7]	f	∅
NO-CONN	**[1]**	[]	**2**	[]	**[2 . . 7]**	f	∅
NO-CONN	**[1, 2]**	[]	**3**	[]	**[3 . . 7]**	f	∅
NO-CONN	**[1 . . 3]**	[]	**4**	[]	**[4 . . 7]**	f	∅
NEW-CONN	[1 . . 3]	[]	4	[]	[4 . . 7]	**t**	{**because$_4$ (•, •)**}
LEFT-ARC$_{\text{Effect}}$	**[1, 2]**	**[3]**	4	[]	[4 . . 7]	t	{because$_4$ (**moved$_3$**, •)}
LEFT-ARC$_{\text{Effect}}$	[1]	**[2, 3]**	4	[]	[4 . . 7]	t	{because$_4$ (**they$_2$** moved$_3$, •)}
NO-ARC-LEFT	**[]**	**[1 . . 3]**	4	[]	[4 . . 7]	t	{because$_4$ (they$_2$ moved$_3$, •)}
NO-ARC-RIGHT	[]	[1 . . 3]	4	**[4]**	**[5 . . 7]**	t	{because$_4$ (they$_2$ moved$_3$, •)}
CONN-FRAG	[]	[1 . . 3]	4	**[4, 5]**	**[6, 7]**	t	{because$_4$**/of$_5$** (they$_2$ moved$_3$, •)}
RIGHT-ARC$_{\text{Cause}}$	[]	[1 . . 3]	4	**[4 . . 6]**	**[7]**	t	{because$_4$/of$_5$ (they$_2$ moved$_3$, **the$_6$**)}
RIGHT-ARC$_{\text{Cause}}$	[]	[1 . . 3]	4	**[4 . . 7]**	**[]**	t	{because$_4$/of$_5$ (they$_2$ moved$_3$, the$_6$ **schools$_7$**)}
SHIFT	**[1 . . 4]**	**[]**	**5**	**[]**	**[5 . . 7]**	**f**	{because$_4$/of$_5$ (they$_2$ moved$_3$, the$_6$ schools$_7$)}
NO-CONN	**[1 . . 5]**	[]	**6**	[]	**[6, 7]**	f	{because$_4$/of$_5$ (they$_2$ moved$_3$, the$_6$ schools$_7$)}
NO-CONN	**[1 . . 6]**	[]	**7**	[]	**[7]**	f	{because$_4$/of$_5$ (they$_2$ moved$_3$, the$_6$ schools$_7$)}
NO-CONN	–	–	–	–	–	–	–

Table 3: The sequence of oracle transitions and states for *Well*$_1$, *they*$_2$ *moved*$_3$ *because*$_4$ *of*$_5$ *the*$_6$ *schools*$_7$. Elements altered by the transition are bolded. Causal language instances are notated as connective(Cause, Effect).

are listed in the supplementary material (§A.3).

4.1 SPLIT transitions

In BECAUSE, a word from one connective can also be part of another connective. This most often occurs with conjoined arguments where portions of the connective are repeated. For example, in *it'll take luck **for** us **to** succeed or even **to** survive, succeed* and *survive* are considered Effects of two different causal instances whose connectives share the *for*. The SPLIT action handles such cases by completing the current causal language instance and starting a new one, copying all connective and argument words up to the repeated connective word.

4.2 Differences from Choi and Palmer

Our scheme differs fourfold from Choi and Palmer:

1. They assume oracle PropBank predicates. DeepCx, lacking oracle connectives, starts new causal language instances with NEW-CONNs, and adds s to the state to track whether such a transition has occurred.
2. Unlike PropBank, BECAUSE allows a connective to include multiple content words. Our system therefore adds a CONN-FRAG transition.
3. A connective word can be part of an argument—e.g., in *enough food to live,* the connective and Cause both include *enough*. DeepCx therefore

compares each connective anchor with itself. (This is why for each new connective anchor a, λ_4 starts out with a as its first element. It is also why the CONN-FRAG action does not advance to the next potential argument word: a connective fragment can be part of an argument.)

4. PropBank never posits two predicates for a single verb, but in BECAUSE, multiple connectives can share a connective word. This case is handled by the new SPLIT transition (see §4.1).

5 DeepCx neural network architecture

Given the experience of previous shallow semantic parsers (e.g., Roth, 2016), we expected performance to depend heavily on syntactic information. We therefore built our system on top of Dyer et al.'s LSTM parser, allowing us to directly incorporate the parser's embeddings. For example, a token's embedding can incorporate the parser's internal embedding of the subtree rooted at that token.

At each step, the network computes a high-dimensional state vector summarizing the internal data structures. That state feeds into a k-dimensional output layer, where k is the number of transition types seen in training. Each vector component is the predicted log probability that the corresponding transition should come next. At test time, the highest-scoring predicted action is taken; in training, gold-standard actions are executed instead.

Figure 1 shows a schematic of the neural network structure. We elaborate on its components below.

5.1 Final state and prediction layers

Beyond λ_{1-4}, the inputs to the state vector are:

- h, the history of actions so far for the sentence.
- d, the path in the dependency parse tree between anchor a and the token being compared with it.
- The lists of tokens making up the connective (o), Cause (c), Effect (e), and Means (m) spans for the causal instance currently under construction.

The parser state $\mathbf{s}$ at each timestep is defined as:

$$\mathbf{s} = \max\{\mathbf{0}, \mathbf{W}_s [\boldsymbol{\lambda_1}; \boldsymbol{\lambda_2}; \mathbf{a}; \boldsymbol{\lambda_3}; \boldsymbol{\lambda_4}; \mathbf{o}; \mathbf{c}; \mathbf{e}; \mathbf{m}; \mathbf{d}; \mathbf{h}] + \mathbf{b}_s\},$$

where $\mathbf{b}_s$ is a bias term, $\mathbf{W}$ is a learned parameter matrix, and any other bold variable $\mathbf{x}$ indicates an embedding of a variable x (described in §5.2). max indicates a component-wise ReLU.

The predicted probability of each transition T is computed from $\mathbf{s}$ using a softmax unit:

$$p(T \mid \mathbf{s}) = \exp\left(\mathbf{g}_T^\top \mathbf{s} + q_T\right)/z,$$

where $\mathbf{g}_T$ is a learned embedding of T, q_T is a bias for T, and z is a normalizing constant.

5.2 Embedding the inputs to the state

5.2.1 Embedding a token

Following Dyer et al. (2015), each token t is represented as a concatenation of three vector inputs:

- $\tilde{\mathbf{w}}_t$, a fixed word embedding for t's surface form.
- $\mathbf{w}_t$, a small additional word embedding of t, which allows the network to learn task-specific representations of words related to causality. This is the only component of a token's representation that is trained specifically for this task (i.e., that does not use an embedding from a pretrained language model or a syntactic parsing model).
- $\mathbf{p}_t$, the LSTM parser's internal embedding of the POS tag it assigned to t in preprocessing.

The concatenation is passed through a linear transformation $\mathbf{V}$ (with a bias $\mathbf{b}_t$) and a ReLU:

$$\mathbf{t} = \max\{\mathbf{0}, \mathbf{V}[\tilde{\mathbf{w}}_t; \mathbf{w}_t; \mathbf{p}_t] + \mathbf{b}_t\}$$

5.2.2 Embedding a list of tokens

For each input to the final state vector that is a list of tokens, we add an LSTM cell to the network.

For the spans of the instance under construction—i.e., the connective, Cause, Effect, and Means spans—embedding token lists is straightforward: whenever a transition adds a token to one of these lists, that token's embedding is added to the corresponding LSTM's input sequence. The LSTM's updated output is then used for all subsequent actions until another transition modifies the span.

The procedure for embedding the λ's is more involved. As transitions are taken, tokens may need to be moved between lists—e.g., the argument token is moved from λ_1 to λ_2 after a LEFT-ARC transition, and the connective anchor token is moved from λ_4 to λ_1 on a NO-CONN.

We implement these transfers using stack LSTMs. Initially, all tokens' embeddings are input to λ_4, but in **reverse order**, so that the leftmost token is added last. Then, whenever λ_4's leftmost token t is to be moved—i.e., on a SHIFT, NO-CONN,

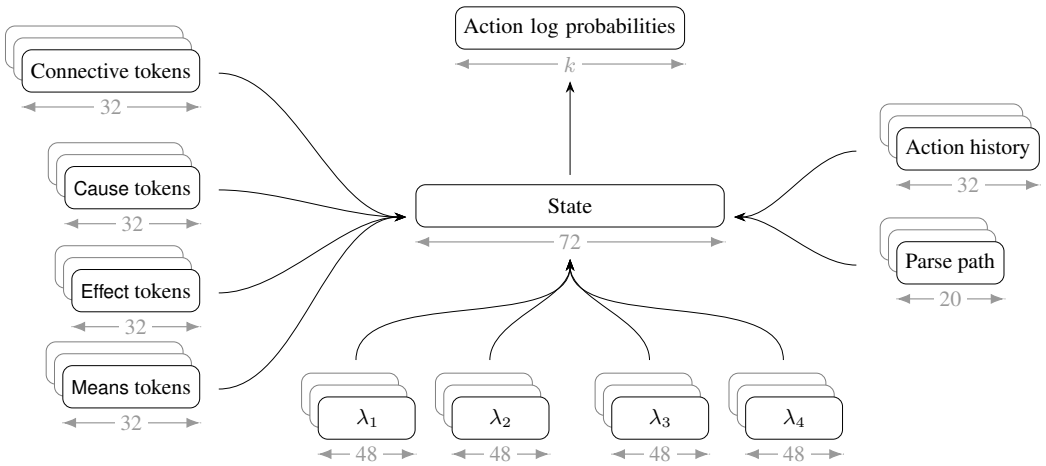


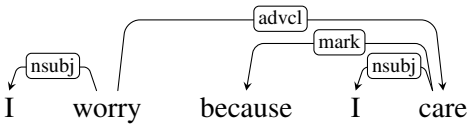
Figure 1: Schematic of the overall neural network architecture. Each lone box represents a vector. Stacked boxes represent LSTMs: at any given time, the state is a single vector, but that state encodes a series of inputs.

RIGHT-ARC, or NO-ARC-RIGHT—the λ_4 LSTM is rewound one step to its state before t was added. Storing λ_4 in reverse order offers the added benefit of tokens closer to the anchor holding greater sway, since LSTMs favor recently added inputs.

λ_1 and λ_2 are a mirror image of λ_4 and λ_3, respectively. Tokens are added to λ_1 on either a SHIFT or a NO-CONN. Thus, the λ_1 LSTM ends up representing an in-order list of tokens up to the current a. If a is then flagged as a connective anchor, tokens to its left are moved from λ_1 to λ_2 as they are compared. The rightmost token t in λ_1 is the first to be compared, so the λ_1 LSTM is rewound to remove t. t's embedding is then added to λ_2, leaving λ_2 with a reversed list of compared tokens.

5.2.3 Embedding a dependency path

The syntactic relationship between the connective anchor a and a candidate argument word t is given to the network as a DEPENDENCY PATH—the series of labels on the dependency arcs between a and t. For instance, consider the dependency parse below:



Here, the path from the first *I* to *because* would be $\bigcirc \xleftarrow{\text{nsubj}} \bigcirc \xrightarrow{\text{advcl}} \bigcirc \xrightarrow{\text{mark}} \bigcirc$, where the blank nodes take the place of the words *I, worry, care,* and *because* in the dependency graph.

To embed a dependency path, we again use the output of an LSTM cell, where each input is an embedding of a dependency label: for a label x, we directly use the LSTM parser's embedding for the **syntactic** parse action LEFT-ARC(x), if available, or RIGHT-ARC(x) otherwise. We add one extra bit to each arc's embedding to indicate whether it was traversed forward or backward in this path.[1]

5.2.4 Embedding the action history

During training, DeepCx learns vector representations of each action. To embed the action history, these action embeddings are fed as inputs into yet another LSTM cell. This LSTM's output is the embedding of the history thus far.

5.3 Implementation details

DeepCx is implemented using a refactored version of the LSTM parser codebase that performs identically to the original.[2] The neural network framework, which also underlies the LSTM parser, is an early version of DyNet (Neubig et al., 2017). The LSTM parser model is pretrained on the usual Penn Treebank (Marcus et al., 1994) sections (training: 02–21; development: 22).

For $\tilde{\mathbf{w}}$, we use the same "structured skip n-gram" word embeddings as the LSTM parser. See Dyer et al. (2015) for details about the embedding approach, hyperparameters, and training corpora. DeepCx gives no special treatment to out-of-vocabulary items, other than using the $\mathbf{0}$ vector for words not included in the pretrained embeddings.

[1] This embedding is similar to that proposed by Roth and Lapata (2016). However, their dependency paths include the words encountered along the way and their POS tags. We experimented with adding these elements to our dependency paths, but found that they consistently reduced performance.

[2] https://github.com/clab/lstm-parser/tree/easy-to-use.

The code for DeepCx is available on GitHub.[3]

5.3.1 Dimensionalities

The pretrained LSTM parser model uses the same dimensionalities as the original LSTM parser.

Token embeddings are 48-dimensional; $\mathbf{w}$ is 10-dimensional. The remaining DeepCx neural network dimensionalities used in the experiments reported below are shown in Figure 1. All LSTM cells use two layers of LSTMs before the final output. These values were chosen as an intuitive balance between values that worked well for other projects and what we could reasonably expect to train with the amount of data we have. Early experiments showed little sensitivity to dimensionality.

6 Experiments

6.1 Experimental setup and training setup

Due to the small corpus size, all experiments use 20-fold cross-validation, split by sentence. Within each fold, the available data—i.e., everything but the fold's held-out test set—is randomized, then split into 80% training and 20% development. After each sentence has been fed through the network, taking gold-standard transitions (see §5), backpropagation is run on all predictions for the sentence. Development set performance is evaluated every 2500 sentences. After each epoch, the training and development sets are re-randomized and re-split.[4] Training ends when either the connective-level F_1 score[5] on the development data hits 0.999 or 85% of the past five epochs' evaluations have yielded lower scores than their immediate predecessors. All systems used the same folds. See the supplementary materials (§A.4) for training parameters.

6.2 Network variants tested in experiments

Ablation studies In addition to the vanilla configuration described above, we examined which non-essential model components contribute to performance. We were particularly interested in the effects of parse information. We tested eliminating the following components of the DeepCx model: (1) $\mathbf{w}$, the task-specific word embeddings, which could contribute to overfitting; (2) $\mathbf{a}$, the action history; and (3) $\mathbf{d}$, the parse path between the connective anchor and the current comparison token.

Argument identification alone DeepCx has no separate argument tagging phase, so we tested performance on the subtask of argument identification by providing DeepCx with oracle transitions only for actions that act on the connective—i.e., NO-CONN, NEW-CONN, CONN-FRAG, and SPLIT. The system was then responsible for deciding between NO-ARC, LEFT-ARC, and RIGHT-ARC transitions.

Restricting generalization One of the strengths of the transition-based approach is its ability to recognize previously unseen forms of causal language that resemble known connectives semantically and/or linguistically. Given our relatively small dataset, however, it seemed possible that the system would not have enough data to make meaningful generalizations. We therefore tested a variant where DeepCx would refuse to allow a test-time NEW-CONN or CONN-FRAG transition unless adding the putative connective word would match the initial word sequence of some connective seen in training.

6.3 Evaluation metrics

For connective discovery we measure precision, recall, and F_1, requiring connectives to match exactly. For argument ID, we split metrics for Causes and Effects (we omit Means, as there are too few in the corpus to evaluate reliably). For each argument type, we report F_1 of connective/argument pairs, where matches must match exactly; F_1 of connective/argument pairs, where half of the larger span's tokens must match; and the average Jaccard index for gold vs. predicted spans, **given a correct connective**. Punctuation is excluded from evaluation.

Jaccard indices convey how close argument tagging is when it does not match exactly. This metric is computed only over true positive connectives, as argument overlap cannot be evaluated automatically for false positives. Thus, Jaccard indices are **not directly comparable** between systems—they represent how well argument ID works given the previous stage, rather than in an absolute sense.

7 Results and analysis

Results are shown in Table 4. For comparison, we also report on the best Causeway configurations.

All significance tests below are paired, two-tailed t-tests on the results from all 20 folds.

[3] https://github.com/duncanka/lstm-causality-tagger.

[4] Reusing the development data means the network can end up memorizing. However, early experiments with dedicated development data showed lower scores, presumably because too much training data was lost from each fold. Of course, our final evaluation is still performed on the fold's held-out data.

[5] For the experiment with oracle connectives, action-level prediction accuracy is used instead of F_1 score.

System variant	Connectives P	R	F_1	Causes F_1	F_1@.5	J	Effects F_1	F_1@.5	J
Best Causeway-S	62.8	46.2	*53.1*	37.9	42.5	81.0	24.8	38.7	73.3
Best Causeway-L	63.4	45.1	52.5	*38.8*	*43.5*	83.7	*30.4*	*40.7*	78.4
Vanilla DeepCx	*63.4*	55.8	*59.2*	*43.9*	*50.6*	83.0	41.0	*51.7*	82.2
No **w**	62.7	56.2	59.1	42.8	49.0	81.9	*41.8*	*51.7*	82.5
No **d**	62.7	***56.5***	*59.2*	42.7	49.1	80.9	39.6	51.1	80.5
No **a**	61.3	54.1	57.3	40.3	48.3	81.6	36.9	50.4	81.1
No novel connectives	65.4	**56.5**	**60.5**	**44.7**	***51.0***	82.6	**42.8**	***52.6***	82.1
Oracle connectives	–	–	–	73.5	80.9	79.7	67.8	82.8	80.7

Table 4: Results for all variants of DeepCx tested. As before, J indicates Jaccard index. For $P/R/F_1$ scores, the best non-oracle results are bolded, and the best results within each of the top two sections are italicized.

7.1 Overall performance

The results show the DeepCx transition system to be a promising approach for SCL.

The vanilla configuration unmistakably eclipses Causeway at connective discovery with a margin of 6.1 F_1 points, driven primarily by recall. Both F_1 scores have high standard deviations across folds (3.6–4.7 points), but the scores covary; some folds are simply harder. DeepCx usually leads Causeway by at least 5 points, making the difference highly statistically significant ($p \ll 0.001$). The gap comes primarily from recall, where DeepCx averages 9.6–10.7 points higher than Causeway. [6]

On end-to-end argument identification, DeepCx again outperforms Causeway, particularly on recall, with a 5–6-point gap in F_1. The Jaccard indices for Causes and Effects are in the low 80's, indicating extensive overlap with gold-standard spans. They are on par with Causeway for Causes and higher for Effects, despite the fact that DeepCx's higher recall gives it more chances to be docked for mismatches.

7.2 Argument identification alone

Argument ID scores remain high when oracle connectives are provided. Naturally, the end-to-end argument scores improve dramatically compared to non-oracle connectives, but the more important question is what fraction of the previous errors remain when connective discovery is no longer a source of error. With oracle connectives, DeepCx achieves 73.5% F_1 on Causes and 67.8% on Effects, implying that the vanilla configuration's argument error was split roughly half and half between connective discovery failures and argument ID failures.

However, the F_1 metrics reflect exact span matches; it is counted as a mismatch if even a single word is off. Because in this experiment the system's entire task is to tag arguments, the Jaccard indices give an absolute measure of overlap between predicted and gold argument spans. By that measure, the neural network's treatment of argument identification transitions looks quite robust. Jaccard indices do drop by a few points compared to non-oracle connectives, as expected: with the oracle, arguments are evaluated for every gold-standard instance, including more difficult ones that the vanilla configuration misses. But despite the more exhaustive assessment, DeepCx maintains Jaccard indices of ∼80% for Causes and Effects.

7.3 Model ablation studies

No pieces of the model beyond the bare essentials improved connective scores. Removing these components did marginally lower argument ID scores, but few differences were statistically significant.

The meager effects of parse paths came as a surprise; indeed, our reason for building on the LSTM parser was to lean on its parse embeddings. That these paths made little difference suggests that the bulk of the information they provide is available in some isomorphic form from simpler inputs.

7.4 Constraining to known connectives

Constraining DeepCx to known connectives yields an interesting tradeoff. On the one hand, it boosts precision ($p < 0.036$) and raises F_1 slightly ($p < 0.09$). Inspecting the vanilla system's outputs accentuates the risks of letting it run wild inventing connectives: its odder proposals included *an unfair effort to, is insanity, eight,* and the dollar sign.

On the other hand, some generalizations were surprisingly perceptive. For instance, the phrase *allowing states greater opportunity to regulate* was not marked by annotators because *allowing* here seems to mean "providing." But DeepCx proposed

[6] Recall should perhaps have even been higher: in at least three cases, DeepCx was penalized for correctly flagging connectives that had been missed by annotators.

allowing opportunity to as a connective—a plausible candidate for annotation. Elsewhere DeepCx tagged *catalyst for* and *fuel* (as in *fueled skepticism*), both arguably annotator omissions.

Ultimately, then, whether to permit novel connectives depends on the user's prioritization of precision, recall, and discovery.

8 What's needed for other constructions and domains?

Although the DeepCx transitions were designed for BECAUSE, it would be straightforward, given appropriate corpora, to extend the transition scheme and model structure to arbitrary frames and role labels as in PropBank and FrameNet. The scheme's arc transitions would need variants for each possible role type, as is standard in existing transition-based SSP (e.g., SLING, Choi and Palmer). Likewise, NEW-CONN could be changed to NEW-CONN(frame); the space of arc transitions for constructing the rest of that instance could then be pruned to those relevant to the frame. As for the tagger state, there are several straightforward ways to modify it for open-ended role and frame labels. One option is to represent each instance's arguments as a list of ⟨*role label*, *list of tokens*⟩ tuples, and to add a frame label variable that is embedded as part of the state. Alternatively, we could follow SLING in providing the tagger a list of ⟨*frame label*, *role label*, *token*⟩ tuples.

Applying SCL to domains beyond causality would be particularly useful for relations like comparison and concession (see Table 1), where complex constructions abound. But as Fillmore et al. (2012) observe, many frames possess the odd non-lexical-unit trigger. For example, the Motion frame can be evoked by the "verb-way" construction (*sang our way across Europe*), and Measurement by the abstract pattern ⟨*number*⟩ ⟨*unit*⟩ ⟨*noun*⟩ (as in *twelve-inch-thick*). Expanding SSP to cover constructions would allow parsing these cases, which are individually rare but collectively form a fat tail of frame instances.

DeepCx already covers most constructional quirks that interfere with SSP, including discontinuous trigger and argument spans, overlaps between arguments, overlaps between trigger words and arguments, and overlaps between triggers. Still, several extensions might be needed for the full gamut of arbitrary constructions. Most notably, our scheme operates on words, but plenty of constructions are sub-lexical (e.g., the comparative *-er*). One solution would be to operate on morphemes instead. Unfortunately, tagging would then be subject to errors in morphological analysis, and morpheme- or character-based embeddings would be needed. A simpler but less elegant solution would be to tag the entire word containing the morpheme (e.g., *bigger*) as part of the construction.

A second challenge is constructions with no lexical trigger, as in *I can't come; I have rehearsal*. The simplest fix would be to add a JUXT transition as a sibling of NEW-CONN. This transition would anchor a new relation instance at the boundary between the words currently being compared, indicating that the mere juxtaposition of two argument spans conveys a relation between them.

Cross-sentential constructions—e.g., discourse connectives whose arguments can be in another sentence—pose a third challenge: our sentence-oriented scheme ignores sentential juxtaposition and cross-sentential grammatical relations as construction possibilities. While it would not be too difficult to alter the scheme to allow, say, arguments in the previous k sentences, it might make randomized training more difficult.

Finally, SPLITs make strong assumptions about how two connectives sharing words will interact. Constructions violating these assumptions may require more drastic surgery on the scheme.

9 Contributions and takeaways

This paper has introduced surface construction labeling as an expansion of shallow semantic parsing. It has also presented DeepCx, a neural transition framework unifying connective discovery and argument ID for causal constructions. DeepCx achieves strong performance on parsing such constructions. Although the transition system targets causal language, its flexibility makes it promising for other domains, as well. We hope DeepCx will inspire further work on SCL. This includes applying more sophisticated tagging techniques such as bidirectional LSTMs, attention, and dynamic oracles, but most importantly developing new data and tasks to which the approach can be applied.

Acknowledgments

The authors thank Miguel Ballesteros, Chris Dyer, Eduard Hovy, Nathan Schneider, and Todd Snider for their invaluable help in developing these ideas, offering feedback, and critiquing draft writeups.

References

Collin Baker, Michael Ellsworth, and Katrin Erk. 2007. SemEval'07 task 19: frame semantic structure extraction. In *Proceedings of the 4th International Workshop on Semantic Evaluations (SemEval 2007)*, pages 99–104. Association for Computational Linguistics.

Timothy Baldwin and Su Nam Kim. 2010. Multiword expressions. In Nitin Indurkhya and Fred J. Damerau, editors, *Handbook of Natural Language Processing*, volume 2, pages 267–292. CRC Press, Boca Raton, USA.

Xavier Carreras and Lluís Màrquez. 2004. Introduction to the CoNLL-2004 shared task: Semantic role labeling. In *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL 2004)*, pages 89–97. Association for Computational Linguistics.

Xavier Carreras and Lluís Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL 2005)*, pages 152–164. Association for Computational Linguistics.

Wenliang Chen, Yue Zhang, and Min Zhang. 2014. Feature embedding for dependency parsing. In *Proceedings of the 25th International Conference on Computational Linguistics: Technical Papers (COLING 2014)*, pages 816–826. Association for Computational Linguistics.

Jinho D. Choi and Martha Palmer. 2011a. Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (HLT '11)*, volume 2, pages 687–692. Association for Computational Linguistics.

Jinho D Choi and Martha Palmer. 2011b. Transition-based semantic role labeling using predicate argument clustering. In *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*, pages 37–45. Association for Computational Linguistics.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537.

Dipanjan Das, Desai Chen, André FT Martins, Nathan Schneider, and Noah A Smith. 2014. Frame-semantic parsing. *Computational Linguistics*, 40(1):9–56.

Jesse Dunietz, Lori Levin, and Jaime Carbonell. 2017a. Automatically tagging constructions of causation and their slot-fillers. In *Transactions of the Association for Computational Linguistics*, volume 5, pages 117–133. Association for Computational Linguistics.

Jesse Dunietz, Lori Levin, and Jaime Carbonell. 2017b. The BECauSE corpus 2.0: Annotating causality and overlapping relations. In *Proceedings of the 11th Linguistic Annotation Workshop (LAW XI)*, pages 95–104. Association for Computational Linguistics.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL 2015)*, pages 334–343. Association for Computational Linguistics.

Charles J. Fillmore, Paul Kay, and Mary Catherine O'Connor. 1988. Regularity and idiomaticity in grammatical constructions: The case of *let alone*. *Language*, 64(3):501–538.

Charles J. Fillmore, Russell Lee-Goldman, and Russell Rhodes. 2012. The FrameNet constructicon. *Sign-Based Construction Grammar*, pages 309–372.

Nicholas FitzGerald, Oscar Täckström, Kuzman Ganchev, and Dipanjan Das. 2015. Semantic role labeling with neural network factors. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP 2015)*, pages 17–21. Association for Computational Linguistics.

William Foland and James Martin. 2015. Dependency-based semantic role labeling using convolutional neural networks. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*, pages 279–288. Association for Computational Linguistics.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. Proceedings of Machine Learning Research.

Adele Goldberg. 1995. *Constructions: A Construction Grammar Approach to Argument Structure*. Chicago University Press.

Adele E. Goldberg. 2013. Constructionist approaches. In *The Oxford Handbook of Construction Grammar*, Oxford Handbooks in Linguistics, pages 15–31. Oxford University Press USA.

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 SIGNLL Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 1–18. Association for Computational Linguistics.

James Henderson, Paola Merlo, Gabriele Musillo, and Ivan Titov. 2008. A latent variable model of synchronous parsing for syntactic and semantic dependencies. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning (CoNLL 2008)*, pages 178–182. Association for Computational Linguistics.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for ucca. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1127–1138. Association for Computational Linguistics.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60. Association for Computational Linguistics.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology (HLT '94)*, pages 114–119. 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:1701.03980.

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

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Michael Ringgaard, Rahul Gupta, and Fernando C. N. Pereira. 2017. SLING: a framework for frame semantic parsing. ArXiv:1710.07032.

Michael Roth. 2016. Improving frame semantic parsing via dependency path embeddings. In *Book of Abstracts of the 9th International Conference on Construction Grammar*, pages 165–167.

Michael Roth and Mirella Lapata. 2016. Neural semantic role labeling with dependency path embeddings. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*, pages 1192–1202. Association for Computational Linguistics.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. 2008. The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning (CoNLL 2008)*, pages 159–177. Association for Computational Linguistics.

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)*, pages 187–197. Association for Computational Linguistics.

Oscar Täckström, Kuzman Ganchev, and Dipanjan Das. 2015. Efficient inference and structured learning for semantic role labeling. *Transactions of the Association for Computational Linguistics*, 3:29–41.

Ivan Titov, James Henderson, Paola Merlo, and Gabriele Musillo. 2009. Online graph planarisation for synchronous parsing of semantic and syntactic dependencies. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence (IJCAI '09)*, pages 1562–1567, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

David Vilares and Carlos Gómez-Rodríguez. 2018. A transition-based algorithm for unrestricted amr parsing. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 142–149. Association for Computational Linguistics.

A Supplementary Material

A.1 Edge case handling: overlapping arguments

Occasionally, a word may be part of multiple arguments of the same connective (e.g., both the Cause and the Effect). For example, in *This equipment is newer and thus safer,* the Cause and Effect of *thus* would respectively be annotated as *this equipment is newer* and *this equipment is safer.* When processing such a shared word, the DeepCx algorithm issues an arc transition that includes both argument names (e.g., LEFT-ARC$_{\text{Cause,Means}}$). From the tagger's standpoint, this is an entirely separate transition from, say, LEFT-ARC$_{\text{Cause}}$ or LEFT-ARC$_{\text{Means}}$.

When executing an action with multiple argument types, a separate arc is added to A for each argument type—i.e., the word is added to both argument spans.

A.2 Parser details

The LSTM parser assumes its input has already been split into sentences and POS-tagged. These preprocessing steps are performed using Stanford CoreNLP (Manning et al., 2014).

A.3 Constraints on transition ordering

As mentioned in §4, several transitions have constraints on their ordering to ensure semantic well-formedness. The following constraints apply:

- A CONN-FRAG may not immediately follow a SPLIT or another CONN-FRAG.
- A SPLIT may not immediately follow a CONN-FRAG or another SPLIT.
- A SPLIT is permitted only if the connective currently under construction has at least one fragment—i.e., it contains at least two words.
- NO-CONN is forbidden if s is true, i.e., if a has been determined to be a connective anchor.

These are enforced at each timestep, both at training and test time, by eliminating violating transitions from the tagger's set of available next actions.

A.4 Neural network details and training parameters

Each LSTM is initialized with its own default item whose values are trained parameters.

We followed the training parameters of Dyer et al. (2015): we used gradient descent for parameter optimization, with an initial learning rate of $\eta_0 = 0.1$ and updates of $\eta_t = \eta_0/(1 + 0.8t)$ after each epoch t; we clipped the ℓ_2 norm of the gradients to 5; and we applied an ℓ_2 penalty of 10^{-6} to all weights. We also used Glorot initialization (Glorot and Bengio, 2010) for all parameters. Each fold took about 40 minutes to train on a single core of a 3.10-GHz processor.