# XCS224N Assignment 3 Dependency Parsing

---

**Due Thursday, March 18 at 11:59pm PT.**

### Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs224n-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

### Submission Instructions

**Coding Submission:** Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

### Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

### Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

# 1 Neural Transition-Based Dependency Parsing

In this assignment, you will build a neural dependency parser using PyTorch. You will implement and train the dependency parser. You'll be implementing a neural-network based dependency parser, with the goal of maximizing performance on the UAS (Unlabeled Attachment Score) metric.

This assignment requires PyTorch without `CUDA` installed. GPUs will be necessary in the next two assignments (via `CUDA`), but are not necessary for this assignment.

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between *head* words, and words which modify those heads. Your implementation will be a *transition-based* parser, which incrementally builds up a parse one step at a time. At every step it maintains a *partial parse*, which is represented as follows

- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

Initially, the stack only contains ROOT, the dependencies list is empty, and the buffer contains all words of the sentence in order. At each step, the parser applies a *transition* to the partial parse until its buffer is empty and the stack size is 1. The following transitions can be applied:

- `SHIFT`: removes the first word from the buffer and pushes it onto the stack.
- `LEFT-ARC`: marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack.
- `RIGHT-ARC`: marks the first (most recently added) item on the stack as a dependent of the second item and removes the first item from the stack.

On each step, your parser will decide among the three transitions using a neural network classifier.

(a) [**6 points (Coding)**] Implement the `__init__` and `parse_step` functions in the `PartialParse` class in `src/submission/parser_transitions.py`. This implements the transition mechanics your parser will use.

(b) [**6 points (Coding)**] Our network will predict which transition should be applied next to a partial parse. We could use it to parse a single sentence by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about *batches* of data at a time (i.e., predicting the next transition for any different partial parses simultaneously). We can parse sentences in minibatches with the following algorithm.

**Algorithm 1** Minibatch Dependency Parsing

**Input:** `sentences`, a list of sentences to be parsed and `model`, our model that makes parse decisions

Initialize `partial_parses` as a list of PartialParses, one for each sentence in `sentences`
Initialize `unfinished_parses` as a shallow copy of `partial_parses`
**while** `unfinished_parses` is not empty **do**
  Take the first `batch_size` parses in `unfinished_parses` as a minibatch
  Use the `model` to predict the next transition for each partial parse in the minibatch
  Perform a parse step on each partial parse in the minibatch with its predicted transition
  Remove the completed (empty buffer and stack of size 1) parses from `unfinished_parses`
**end while**

**Return:** The `dependencies` for each (now completed) parse in `partial_parses`.

Implement this algorithm in the `minibatch_parse` function in `src/submission/parser_transitions.py`.

*Note: You will need* `minibatch_parse` *to be correctly implemented to evaluate the model you will build in part (c). However, you do not need it to train the model, so you should be able to complete most of part (c) even if* `minibatch_parse` *is not implemented yet.*

We are now going to train a neural network to predict, given the state of the stack, buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: *A Fast and Accurate Dependency Parser using Neural Networks.*[1] The function extracting these features has been implemented for you in `src/submission/parser_utils.py`. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers $[w_1, w_2, \ldots, w_m]$ where $m$ is the number of features and each $0 \le w_i < |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). First our network looks up an embedding for each word and concatenates them into a single input vector:

$$\mathbf{x} = [\mathbf{E_{w_1}}, ..., \mathbf{E_{w_m}}] \in \mathbb{R}^{dm}$$

where $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ is an embedding matrix with each row $\mathbf{E_w}$ as the vector for a particular word $w$. We then compute our prediction as:

$$\begin{aligned} \mathbf{h} &= \mathrm{ReLU}(\mathbf{xW} + \mathbf{b_1}) \\ \mathbf{l} &= \mathbf{hU} + \mathbf{b_2} \\ \hat{\mathbf{y}} &= \mathrm{softmax}(l) \end{aligned}$$

where $\mathbf{h}$ is referred to as the hidden layer, $\mathbf{l}$ is referred to as the logits, $\hat{\mathbf{y}}$ is referred to as the predictions, and $\mathrm{ReLU}(z) = \max(z, 0)$). We will train the model to minimize cross-entropy loss:

$$J(\theta) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i$$

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

(c) **[9 points (Coding)]** In `src/submission/parser_model.py` you will find skeleton code to implement this simple neural network using PyTorch. Complete the `__init__`, `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` function within the `src/submission/train.py` file.

Finally execute `python run.py` within the `src/` subdirectory to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies). Make sure to turn off debug setting by setting `debug=False` in the `main` function of `run.py`.

**Hints:**

- When debugging, set `debug=True` in the `main` function of `src/run.py`. This will cause the code to run over a small subset of the data, so that training the model won't take as long. Make sure to set `debug=False` to run the full model once you are done debugging.
- When running with `debug=True`, you should be able to get a loss smaller than 0.2 and a UAS larger than 65 on the dev set (although in rare cases your results may be lower, there is some randomness when training).
- It should take about **1 hour** to train the model on the entire the training dataset, i.e., when `debug=False`.
- When running with `debug=False`, you should be able to get a loss smaller than 0.08 on the train set and an Unlabeled Attachment Score larger than 87 on the dev set. For comparison, the model in the original neural dependency parsing paper gets 92.5 UAS. If you want, you can tweak the hyperparameters for your model (hidden layer size, hyperparameters for Adam, number of epochs, etc.) to improve the performance (but you are not required to do so).

[1] Chen and Manning, 2014, https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf

### Deliverables

For this assignment, please submit all files within the `src/submission` subdirectory. This includes:

- `src/submission/__init__.py`
- `src/submission/parser_model.py`
- `src/submission/parser_transitions.py`
- `src/submission/parser_utils.py`
- `src/submission/train.py`

## 2 Quiz

This remainder of this homework is a series of multiple choice questions related to the word2vec algorithm.

**How to submit:** Even though these are not coding questions, you will submit your response to each question in the `src-quiz/submission.py` file. This file will act as your 'bubble sheet' for multiple choice questions in this course. A sample response might look like this:

```
def multiple_choice_1a():
  """"
  # Return a python collection with the option(s) that you believe are correct
  # like this:
  # `return ['a']`
  # or
  # `return ['a', 'd']`
  response = []
  ### START CODE HERE ###
  ### END CODE HERE ###
  return response
```

If you believe that `a` and `b` are the correct responses to this question, you will type `response = ['a', 'b']` between the indicated lines like this:

```
def multiple_choice_1a():
  """"
  # Return a python collection with the option(s) that you believe are correct
  # like this:
  # `return ['a']`
  # or
  # `return ['a', 'd']`
  response = []
  ### START CODE HERE ###
  response = ['a', 'b']
  ### END CODE HERE ###
  return response
```

**How to verify your submission:** You can run the student version of the autograder locally like all coding problem sets. In the case of this problem set, the helper tests will verify that your responses are within the set of possible choices for each question (e.g. the helper functions will flag if you forget to answer a question of if you respond with `['a', 'd']` when the choices are `['a', 'b', 'c']`.) See the front pages of this assignment for instructions to run the autograder.

1. **[2 points]** Recall the standard Stochastic Gradient Descent update rule

$$\theta \leftarrow \theta - \alpha \nabla_\theta J_{\text{minibatch}}(\theta)$$

where $\theta$ is a vector containing all of the model parameters, $J$ is the loss function, $\nabla_\theta J_{\text{minibatch}}(\theta)$ is the gradient of the loss function with respect to the parameters on a minibatch of data, and $\alpha$ is the learning rate.

Adam additionally uses a trick called momentum by keeping track of $m$, a rolling average of the gradients:

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_\theta J_{\text{minibatch}}(\theta)$$

$$\theta \leftarrow \theta - \alpha m$$

where $\beta_1$ is a hyperparameter between 0 and 1 (often set to 0.9). This momentum trick helps in converging faster. Which of the following is true regarding the gradient update using momentum?

(a) Relative to SGD, each update will not vary as much (the current gradient receives only a $1 - \beta 1$ scaled update). This helps maintain a smaller variance and helps in faster convergence to a local optimum.

(b) Relative to SGD, each update has a larger variance. This helps in faster convergence to a local optimum.

(c) Setting $\beta_1$ to a low value would lead to faster convergence to a local optimum, relative to SGD.

2. **[2 points]** Adam uses adaptive learning rates by keeping track of $v$, a rolling average of the magnitudes of the gradient:

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_\theta J_{\text{minibatch}}(\theta)$$

$$v \leftarrow \beta_2 v + (1 - \beta_2)(\nabla_\theta J_{\text{minibatch}}(\theta) \odot \nabla_\theta J_{\text{minibatch}}(\theta)$$

$$\theta \leftarrow \theta - \alpha \odot m / \sqrt{v}$$

where $\odot$ and / denote element-wise multiplication and division (so $z \odot z$ is element-wise squaring) and $\beta_2$ is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{v}$, which of the model parameters will get larger updates?

(a) The parameters with the smaller gradients (on average) will get larger updates. This means that when the loss is more flat with respect to a parameter, that parameter will get a larger update, helping to move off the flat area.

(b) The parameters with the largest gradients (on average) will get larger updates. This means that when the loss is more steep with respect to a parameter, that parameter will get a larger update, helping to move off of the steep area.

(c) None of the above.

Note: Dropout is a regularization technique. If you are unfamiliar with the concept, you can read more in this handout from CS231n.

3. During training, dropout randomly sets units in the hidden layer h to zero and this happens with a probability $p_{\text{drop}}$ (dropping different units each minibatch) and then multiplies h by a constant $\gamma$ We can write this as

$$h_{\text{drop}} = \gamma d \circ h$$

where $d \in 0, 1^{D_k}$ ( where $D_k$ is the size $h$) of is a mask vector where each entry is 0 with probability $p_{\text{drop}}$ and 1 with probability $(1 - p_{\text{drop}})$. $\gamma$ is chosen such that the expected value of $h_{\text{drop}}$ is $h$.

$$\mathbb{E}_{p_{\text{drop}}}[h_{\text{drop}}]_i = h_{\text{i}}$$

For all $i \in 1, ..., D_k$.

For example, let the hidden layer $h$ have 5 nodes and let $p_{\text{drop}}$ be set to 0.6,

$h = [0.33, -1.18, 0.7, -1.8, 0.21]$ (vector representing weights at each node)

$d = [1, 0, 0, 1, 0]$ ($d$ is randomly generated based on the value $p_{\text{drop}}$)

$h_{\text{drop}} = \gamma d\dot{h}$

$h_{\text{drop}} = \gamma[0.33, 0, 0, -1.8, 0]$

3a. **[1 point]** What must $\gamma$ equal in terms of $p_{\text{drop}}$?

(a)
$$\gamma = 1/(p_{\text{drop}})$$

(b)
$$\gamma = 1/(1 - p_{\text{drop}})$$

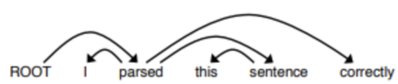
(c)
$$\gamma = (1 - p_{\text{drop}})/(p_{\text{drop}})$$

3b. **[1 point]** Which among the below options are correct regarding dropout at train and test time?

(a) We apply dropout only at train time.

(b) We apply dropout at both train and test time.

4. Work through the sequence of transitions needed for parsing the sentence **"I parsed this sentence correctly"**. The dependency tree for the sentence is shown below. At each step, fill the missing transitions (marked in roman numerals in red) in the configuration (table) of the stack and buffer, as well as what transition was applied at each step and what new dependency was added (if any). A few of the steps are filled in for you.




| Stack | Buffer | New dependency | Transition |
|---|---|---|---|
| [ROOT] | [I, parsed, this, sentence, correctly] | | Initial Configuration |
| [ROOT, I] | [parsed, this, sentence, correctly] | | SHIFT |
| [ROOT, I, parsed] | [this, sentence, correctly] | | SHIFT |
| [ROOT, parsed] | [this, sentence, correctly] | parsed→I | LEFT-ARC |
| [ROOT, parsed, this] | [(i)] | | (ii) |
| [ROOT, parsed, this, sentence] | [correctly] | | SHIFT |
| [ROOT, parsed, sentence] | [correctly] | (iii) | (iv) |
| [ROOT, parsed] | [correctly] | (v) | (vi) |
| [ROOT, parsed, correctly] | [(vii)] | | (viii) |
| [ROOT, parsed] | [] | parsed→correctly | RIGHT-ARC |
| [ROOT] | [] | (ix) | (x) |

4a. **[0.50 points]** Select the right option for blanks (i) and (ii):

(a) (i) : [correctly]; (ii) SHIFT

(b) (i) : [sentence, correctly]; (ii) SHIFT

(c) (i) : parsed → this; (ii) LEFT-ARC

(d) (i) : parsed → this; (ii) RIGHT-ARC

4b. **[0.50 points]** Select the right option for blanks (i) and (ii):

(a) (iii) : leave it blank ; (iv) SHIFT

(b) (iii) : sentence → correctly; (iv) LEFT-ARC

(c) (iii) : sentence → this; (iv) LEFT-ARC

(d) (iii) : sentence → this; (iv) RIGHT-ARC

4c. **[0.50 points]** Select the right option for blanks (v) and (vi):

(a) (v) : leave it blank ; (vi) SHIFT

(b) (v) : sentence → correctly; (vi) LEFT-ARC

(c) (v) : sentence → this; (vi) RIGHT-ARC

(d) (v) : parsed → sentence; (vi) RIGHT-ARC

4d. **[0.50 points]** Select the right option for blanks (vii) and (viii):

(a) (vii) : [correctly] ; (viii) SHIFT

(b) (vii) : []; (viii) : SHIFT

(c) (vii) : parsed → correctly; (viii) LEFT-ARC

(d) (vii) : parsed → correctly; (viii) RIGHT-ARC

4e. **[0.50 points]** Select the right option for blanks (ix) and (x):

(a) (ix) : ROOT → parsed; (x) RIGHT-ARC

(b) (ix) : ROOT → parsed; (x) LEFT-ARC

(c) (ix) : keep it blank; (x) SHIFT

5. **[0.50 points]** A sentence containing $n$ words will be parsed in how many steps (in terms of $n$)?

(a) $2n$ steps

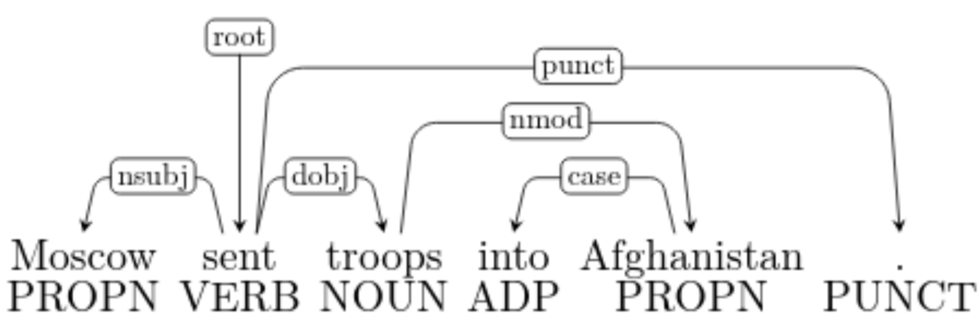
(b) $n$ steps

(c) $n^3$ steps

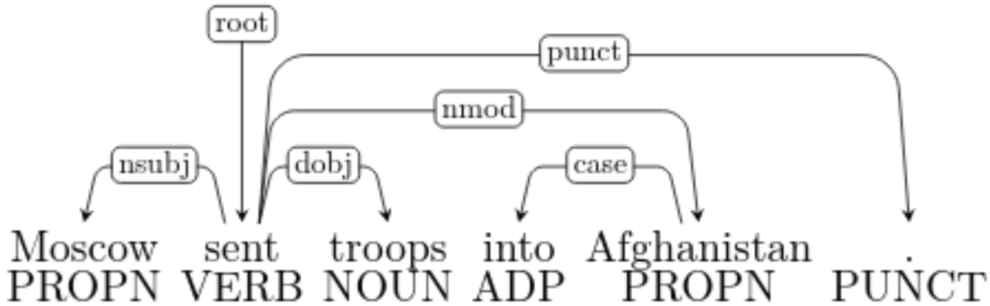
(d) $0.5n$ steps

**Parsing Errors**

We'd like to look at example dependency parses and understand where parsers like ours might be wrong. For example, in this sentence:




the dependency of the phrase **into Afghanistan** is wrong because the phrase should modify **sent** (as in *sent into Afghanistan*) not **troops** (because *troops into Afghanistan* doesn't make sense). Here is the correct parse:




More generally, here are four types of parsing error:

- **Prepositional Phrase Attachment Error:** In the example above, the phrase *into Afghanistan* is a prepositional phrase. A Prepositional Phrase Attachment Error is when a prepositional phrase is attached to the wrong head word (in this example, *troops* is the wrong head word and sent is the correct head word. More examples of prepositional phrases include *with a rock*, *before midnight* and *under the carpet*.
- **Verb Phrase Attachment Error:** In the sentence *Leaving the store unattended, I went outside to watch the parade*, the phrase *leaving the store unattended* is a verb phrase. A Verb Phrase Attachment Error is when a verb phrase is attached to the wrong head word (in this example, the correct head word is *went*).
- **Modifier Attachment Error:** In the sentence, *I am extremely short*, the adverb *extremely* is a modifier of the adjective *short*. A Modifier Attachment Error is when a modifier is attached to the wrong head word (in this example, the correct head word is *short*).
- **Coordination Attachment Error:** In the sentence *Would you like brown rice or garlic naan*? the phrases *brown rice* and *garlic naan* are both conjuncts and the word or is the coordinating conjunction. The second conjunct (here *garlic naan*) should be attached to the first conjunct (here *brown rice*). A Coordination Attachment Error is when the second conjunct is attached to the wrong head word (in this example, the correct head word is *rice*). Other coordinating conjunctions include *and*, *but*, and *so*.

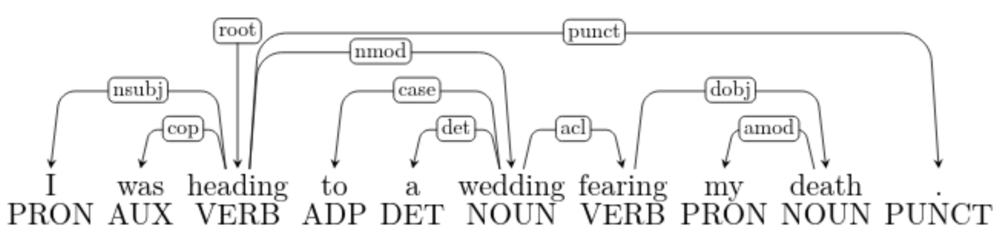
This question presents four sentences with dependency parses obtained from a parser. Each sentence has one error, and there is one example of each of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. To demonstrate: for the example above, you would write:

- **Error type:** Prepositional Phrase Attachment Error
- **Incorrect dependency:** troops → Afghanistan
- **Correct dependency:** sent → Afghanistan

**Note:** *There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website:* `http://universaldependencies.org`*. However, you* **do not** *need to know all these details in order to do these questions. In each of these cases, we are asking about the attachment of phrases and it should be sufficient to see if they are modifying the correct head. In particular, you* **do not** *need to look at the labels on the dependency edges – it suffices to just look at the edges themselves.*

For each sentence, select the correct combination of Error Type, Incorrect Dependency, and Correct Dependency.

6. **[1 point]**



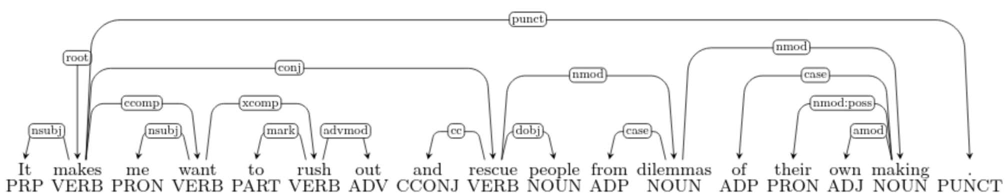
(a) **Error type:** Verb Phrase Attachment Error
**Incorrect dependency:** wedding → fearing
**Correct dependency:** I → fearing

(b) **Error type:** Prepositional Phrase Attachment Error
**Incorrect dependency:** wedding → fearing
**Correct dependency:** I → death

(c) **Error type:** Verb Phrase Attachment Error
**Incorrect dependency:** wedding → fearing
**Correct dependency:** heading → I

(d) **Error type:** Coordination Attachment Error
**Incorrect dependency:** wedding → fearing
**Correct dependency:** heading → death OR I → death

7. [**1 point**]



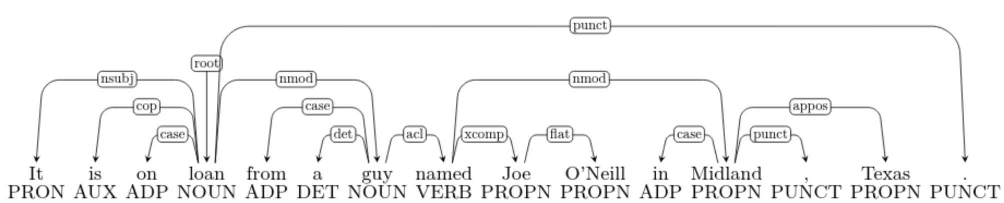

(a) **Error type:** Prepositional Phrase Attachment Error
**Incorrect dependency:** makes → rescue
**Correct dependency:** rush → dilemma

(b) **Error type:** Modifier Attachment Error
**Incorrect dependency:** makes → rescue
**Correct dependency:** want → rescue

(c) **Error type:** Coordination Attachment Error
**Incorrect dependency:** makes → rescue
**Correct dependency:** rush → rescue

(d) **Error type:** Verb Phrase Attachment Error
**Incorrect dependency:** makes → rescue
**Correct dependency:** want → rush

8. [**1 point**]



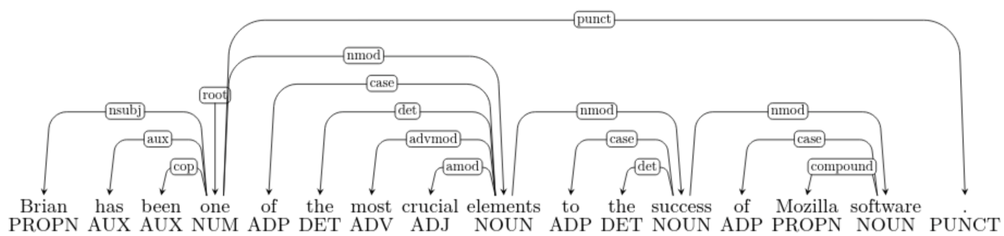

(a) **Error type:** Coordination Attachment Error
**Incorrect dependency:** named → Midland
**Correct dependency:** joe → Midland

(b) **Error type:** Modifier Attachment Error
**Incorrect dependency:** named → Midland
**Correct dependency:** loan → joe

(c) **Error type:** Prepositional Phrase Attachment Error
**Incorrect dependency:** named → Midland
**Correct dependency:** guy → Midland

(d) **Error type:** Verb Phrase Attachment Error
**Incorrect dependency:** named → Midland
**Correct dependency:** load → Midland

9. **[1 point]**




(a) **Error type:** Coordination Attachment Error
**Incorrect dependency:** elements → most
**Correct dependency:** elements → software

(b) **Error type:** Modifier Attachment Error
**Incorrect dependency:** elements → most
**Correct dependency:** crucial → most

(c) **Error type:** Prepositional Phrase Attachment Error
**Incorrect dependency:** elements → most
**Correct dependency:** one → success

(d) **Error type:** Verb Phrase Attachment Error
**Incorrect dependency:** elements → most
**Correct dependency:** one → software

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

---

THERE IS NO WRITTEN SUBMISSION FOR THIS ASSIGNMENT.

YOU ARE NOT REQUIRED TO SUBMIT ANYTHING.