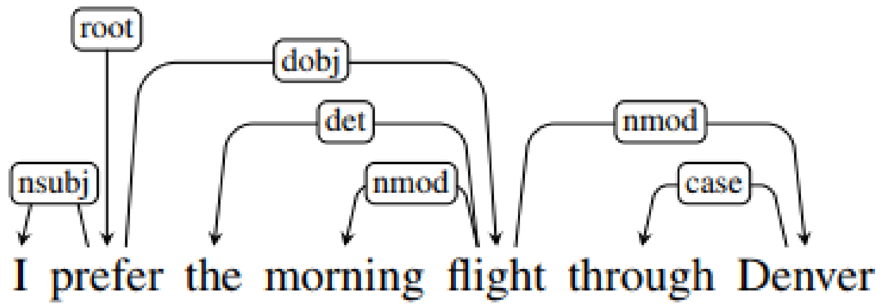
# 8.1 - Dependency Parsing

**CS 6320**

## Dependency Grammars

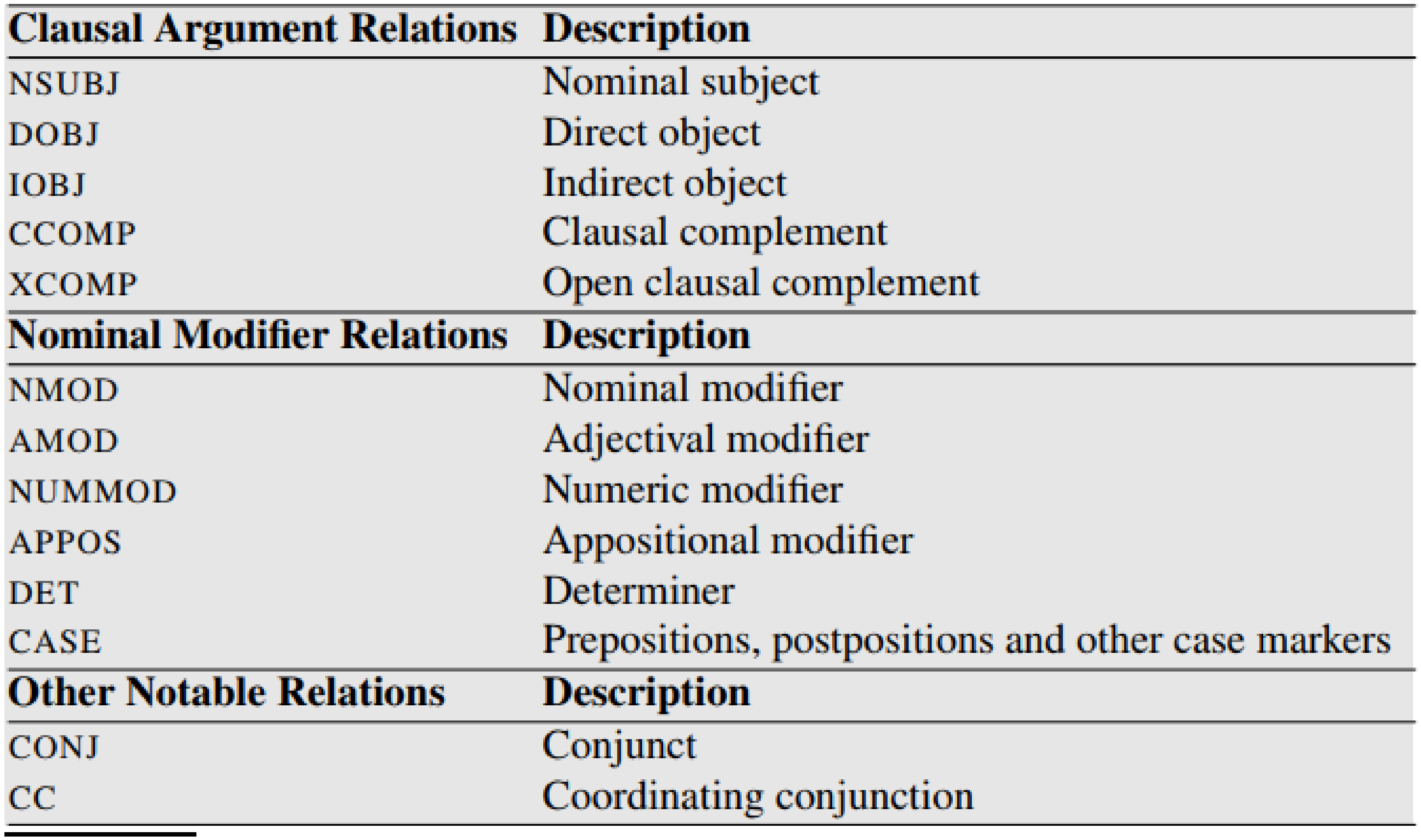
In dependency grammars **phrasal constituents** and **phrase-structure rules** do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of the words (or lemmas) in a sentence and an associated set of **directed binary grammatical relations** that hold among the words.



Relations among the words are illustrated in the above sentence with directed, labeled arcs from heads to dependents. We call this a **typed dependency structure** typed dependency because the labels are drawn from a fixed inventory of grammatical relations. It also includes a root node that explicitly marks the root of the tree, the head of the entire structure.

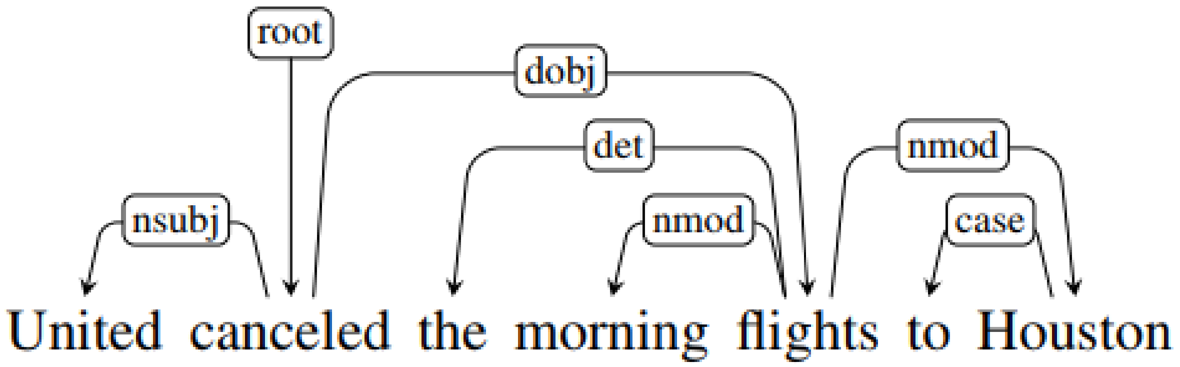
# Selected Dependency Relations from the Universal Dependency Set

- The Universal Dependencies Universal Dependencies project (Nivre et al., 2016) provides an inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically applicable



| Clausal Argument Relations | Description |
|---|---|
| NSUBJ | Nominal subject |
| DOBJ | Direct object |
| IOBJ | Indirect object |
| CCOMP | Clausal complement |
| XCOMP | Open clausal complement |
| **Nominal Modifier Relations** | **Description** |
| NMOD | Nominal modifier |
| AMOD | Adjectival modifier |
| NUMMOD | Numeric modifier |
| APPOS | Appositional modifier |
| DET | Determiner |
| CASE | Prepositions, postpositions and other case markers |
| **Other Notable Relations** | **Description** |
| CONJ | Conjunct |
| CC | Coordinating conjunction |

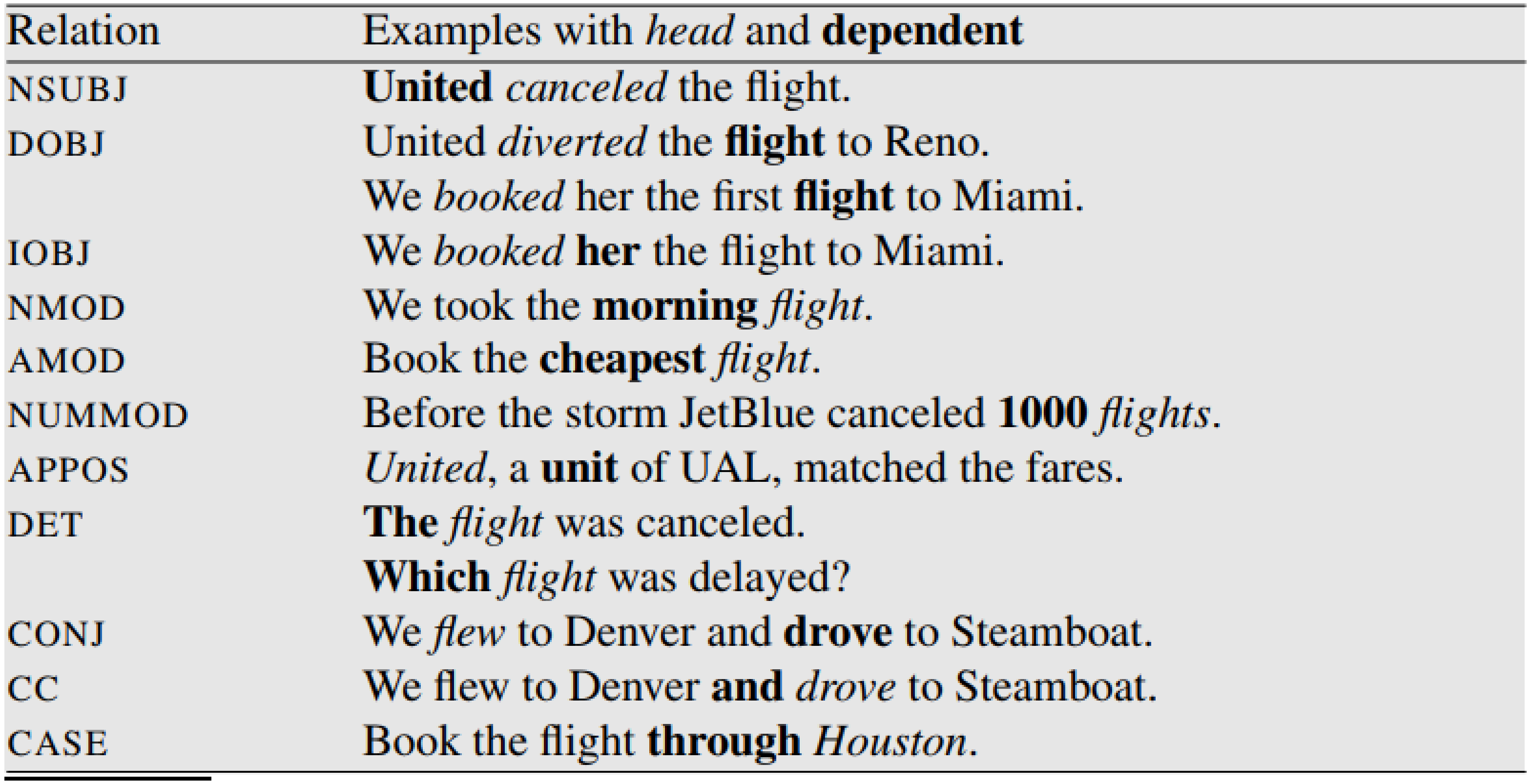
## An Example of Dependency Parse



- The clausal relations NSUBJ and DOBJ identify the subject and direct object of the predicate cancel, while the NMOD, DET, and CASE relations denote modifiers of the nouns flights and Houston.

## More Examples

- Example sentences of core Universal Dependency relations



| Relation | Examples with *head* and **dependent** |
|---|---|
| NSUBJ | **United** *canceled* the flight. |
| DOBJ | United *diverted* the **flight** to Reno. |
| | We *booked* her the first **flight** to Miami. |
| IOBJ | We *booked* **her** the flight to Miami. |
| NMOD | We took the **morning** *flight*. |
| AMOD | Book the **cheapest** *flight*. |
| NUMMOD | Before the storm JetBlue canceled **1000** *flights*. |
| APPOS | *United*, a **unit** of UAL, matched the fares. |
| DET | **The** *flight* was canceled. |
| | **Which** *flight* was delayed? |
| CONJ | We *flew* to Denver and **drove** to Steamboat. |
| CC | We flew to Denver **and** *drove* to Steamboat. |
| CASE | Book the flight **through** *Houston*. |

## A Dependency Tree

- A dependency tree is a directed graph that satisfies the following constraints:
  - There is a single designated root node that has no incoming arcs.
  - With the exception of the root node, each vertex has exactly one incoming arc.
  - There is a unique path from the root node to each vertex in V.

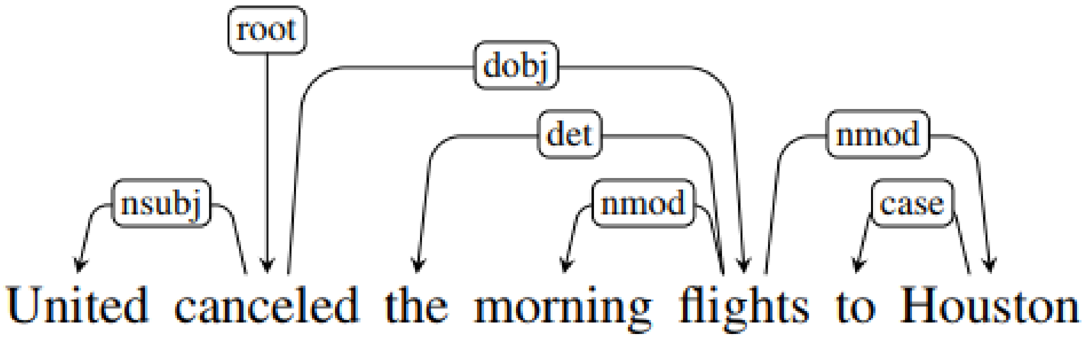
Taken together, these constraints ensure that:

- each word has a single head,
- that the dependency structure is connected, and
- that there is a single root node from which one can follow a unique directed path to each of the words in the sentence.

## Projectivity

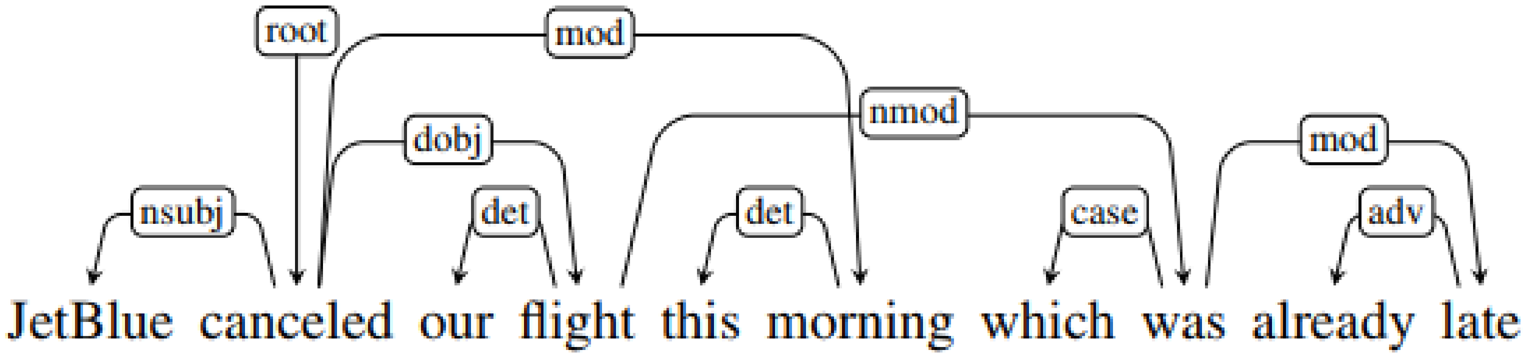
- An arc from a head to a dependent is said to be **projective** if there is a **path** from the head to every word that lies between the head and the dependent in the sentence.

- A dependency tree is said to be projective if all the arcs that make it up are projective (as shown in the example below)



## Projectivity

- There are, however, many perfectly valid constructions which lead to non-projective trees, particularly in languages with a relatively flexible word order.


root
mod
dobj
nmod
mod
nsubj
det
det
case
adv
JetBlue canceled our flight this morning which was already late

- In the above non-projective example, the arc from flight to its modifier was is non-projective since there is no path from flight to the intervening words this and morning.
- This diagram shows that  projectivity (and non-projectivity) can be detected in the way we've been drawing our trees: A dependency tree is projective if it can be drawn with no crossing edges. Here there is no way to link flight to its dependent was without crossing the arc that links morning to its head.

## Dependency Treebanks

- For the most part, directly annotated dependency treebanks have been created for **morphologically rich languages** such as Czech, Hindi and Finnish that lend themselves to dependency grammar approaches, with the Prague Dependency Treebank (Bejcek et al., 2013) ˇfor Czech being the most well-known effort.

- The major English dependency treebanks have largely been extracted from existing resources such as the Wall Street Journal sections of the Penn Treebank(Marcus et al., 1993).

- The more recent **OntoNotes project** (Hovy et al. 2006,Weischedel et al. 2011) extends this approach going beyond traditional news text to include conversational telephone speech, weblogs, usenet newsgroups, broadcast, and talk shows in English, Chinese and Arabic

# Translation from Constituent to Dependency Structures

- has two subtasks:
  - identifying all the head-dependent relations in the structure and
  - identifying the correct dependency relations for these relations.

The first task relies heavily on the use of head rules discussed when we covered constituency parsing, first developed for use in lexicalized probabilistic parsers (Magerman 1994,Collins 1999,Collins 2003).

Here's a simple and effective algorithm from Xia and Palmer (2001):

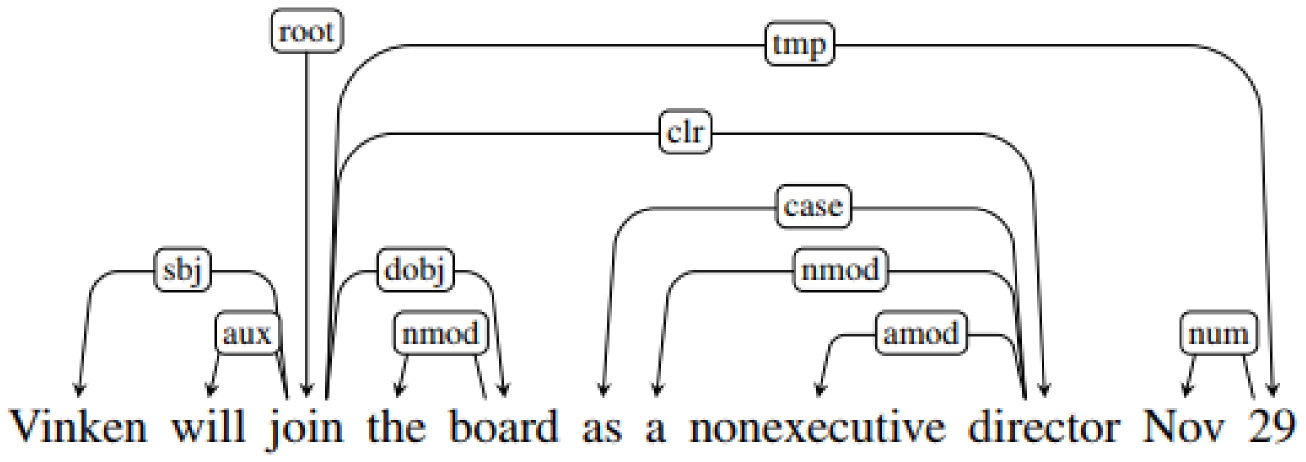
A. Mark the head child of each node in a phrase structure, using the appropriate head rules.

B. In the dependency structure, make the head of each non-head child depend on the head of the head-child.

When a phrase-structure parse contains additional information in the form of grammatical relations and function tags, as in the case of the Penn Treebank, these tags can be used to label the edges in the resulting tree.

## Automatically Generated Dependency Tree

The definitions: https://nlp.stanford.edu/software/dependencies_manual.pdf


root
tmp
clr
case
sbj
dobj
nmod
aux
nmod
amod
num
Vinken will join the board as a nonexecutive director Nov 29

| Clausal Argument Relations | Description |
|---|---|
| NSUBJ | Nominal subject |
| DOBJ | Direct object |
| IOBJ | Indirect object |
| CCOMP | Clausal complement |
| XCOMP | Open clausal complement |
| **Nominal Modifier Relations** | **Description** |
| NMOD | Nominal modifier |
| AMOD | Adjectival modifier |
| NUMMOD | Numeric modifier |
| APPOS | Appositional modifier |
| DET | Determiner |
| CASE | Prepositions, postpositions and other case markers |
| **Other Notable Relations** | **Description** |
| CONJ | Conjunct |
| CC | Coordinating conjunction |

# Transition-Based Dependency Parsing

- Motivated by a stack-based approach called **shift-reduce parsing** originally developed for analyzing programming languages (Aho and Ullman, 1972).

This classic approach is simple and elegant, employing:

- a context-free grammar,
- a stack, and
- a list of tokens to be parsed.

Input tokens are successively shifted onto the stack and

- the top two elements of the stack are matched against the right-hand side of <u>the rules in the grammar</u>;
- when a match is found the matched elements are replaced on the stack (reduced) by the non-terminal from the left-hand side of the rule being matched.

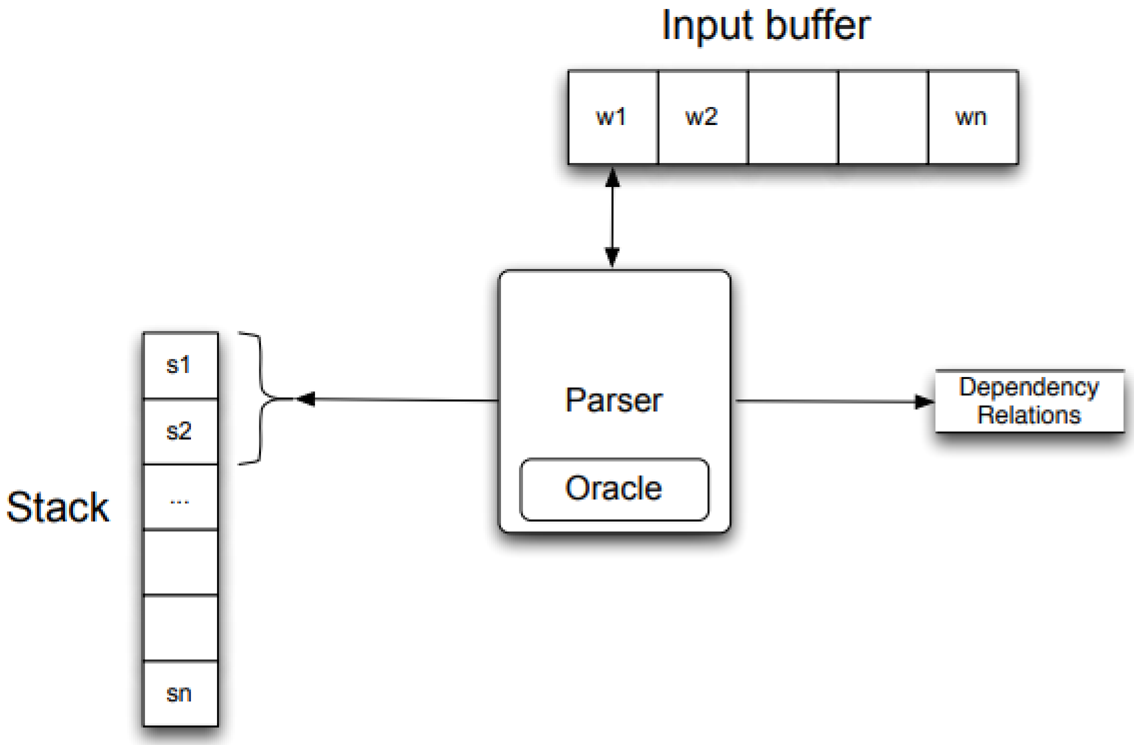
In adapting this approach for dependency parsing, we forgo the explicit use of a grammar and alter the reduce operation so that instead of adding a non-terminal to a parse tree, <u>it introduces a dependency relation between a word and its head</u>.

# The Reduce Action

The reduce action is replaced with two possible actions:

- assert a head-dependent relation between the word at the top of the stack and the word below it, or
- vice versa.

The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration.



# Configurations

- A key element in transition-based parsing is the notion of a configuration which consists of:
  - a stack,
  - an input buffer of words, or tokens, and
  - a set of relations representing a dependency tree.

Given this framework, the parsing process consists of a sequence of transitions through the space of possible configurations.

- **The goal** of this process is to find a final configuration where all the words have been accounted for and an appropriate dependency tree has been synthesized.
- A **search problem**

## The Search

To implement the search, we define **a set of transition operators**, which when applied to a configuration produce new configurations.

- Given this setup, we can view the operation of a parser as a searc through a space of configurations for a sequence of transitions tha leads from a start state to a desired goal state.

At the start of this process we create an initial configuration in which:

- the stack contains the **ROOT** node,
- the word list is initialized with the set of the words or lemmatized tokens in the sentence, and
- an **empty set of relations** is created to represent the parse.

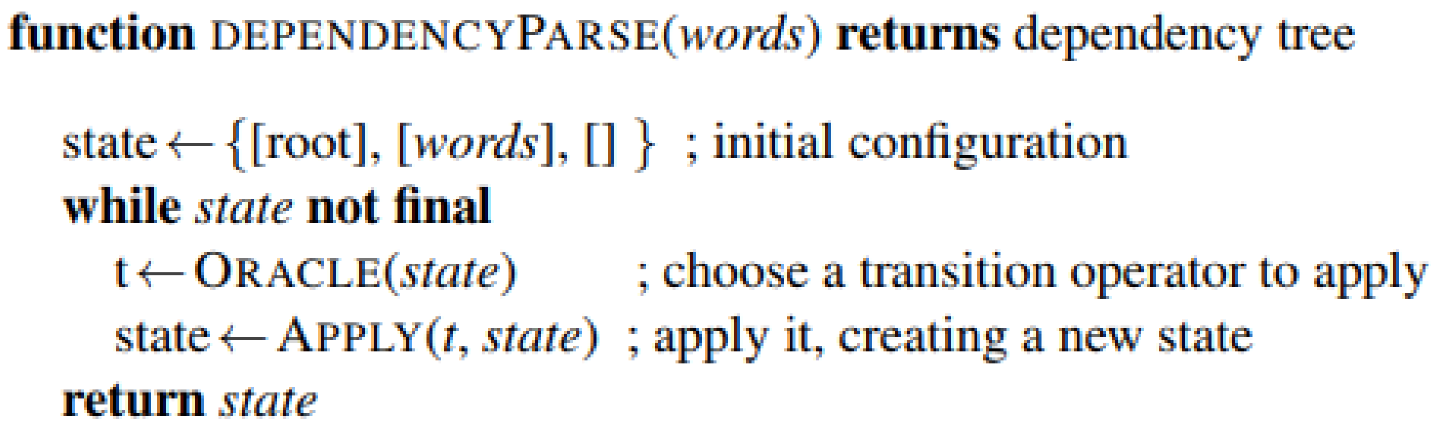
In the final goal state, the stack and the word list should be empty, and the set of relations will represent the final parse.

## The Operators

- In the standard approach to transition-based parsing, the operators used to produce new configurations are surprisingly simple and correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right.
  - Assign the current word as the head of some previously seen word;
  - Assign some previously seen word as the head of the current word;
  - Or postpone doing anything with the current word, adding it to a store for later processing.
- To make these actions more precise, we'll create three transition operators that will operate on the top two elements of the stack:
  A. **LEFTARC**: Assert a head-dependent relation between the word at the top of stack and the word directly beneath it; remove the lower word from the stack.
  B. **RIGHTARC**: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;
  C. **SHIFT:** Remove the word from the front of the input buffer and push it onto the stack.

# Arc Standard Approach to Transition-based Parsing

- To assure that these operators are used properly we'll need to add some preconditions to their use:
  a) Since, by definition, the ROOT node cannot have any incoming arcs, we'll add the restriction that the LEFTARC operator cannot be applied when ROOT is the second element of the stack.
  b) Both reduce operators (LEFTARC and RIGHTARC) require two elements to be on the stack to be applied. Given these transition operators and preconditions, the specification of a transition-based parser is quite simple.



```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
     t ← ORACLE(state)        ; choose a transition operator to apply
     state ← APPLY(t, state)  ; apply it, creating a new state
  return state
```

## Dependency Parsing

NOTE: this is a greedy search algorithm — the oracle provides a single choice at each step and the parser proceeds with that choice, no other options are explored, no backtracking is employed, and a single parse is returned in the end.

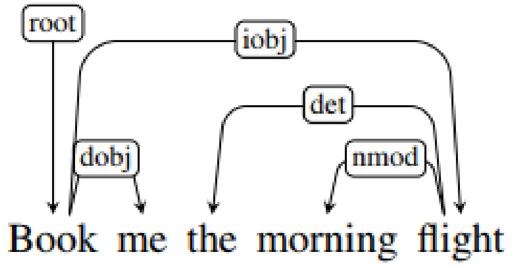
```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)          ; choose a transition operator to apply
    state ← APPLY(t, state)  ; apply it, creating a new state
  return state
```

- At each step, the parser consults an oracle that provides the correct transition operator to use given the current configuration. It then applies that transition operator to the current configuration, producing a new configuration. The process ends when all the words in the sentence have been consumed and the ROOT node is the only element remaining on the stack.

Let us consider the first configuration:

After word "me" has been read:

| Stack | Word List | Relations |
|---|---|---|
| [root, book, me] | [the, morning, flight] | |



The correct operator to apply here is RIGHTARC which assigns "book" as the head of "me" and pops "me" from the stack resulting in the following configuration:

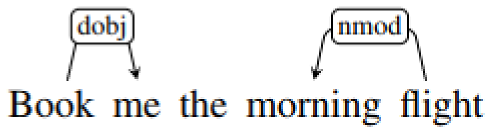
| Stack | Word List | Relations |
|---|---|---|
| [root, book] | [the, morning, flight] | (book $\rightarrow$ me) |

After several subsequent applications of the SHIFT and LEFTARC operators, the configuration after reading all the words looks like the following:

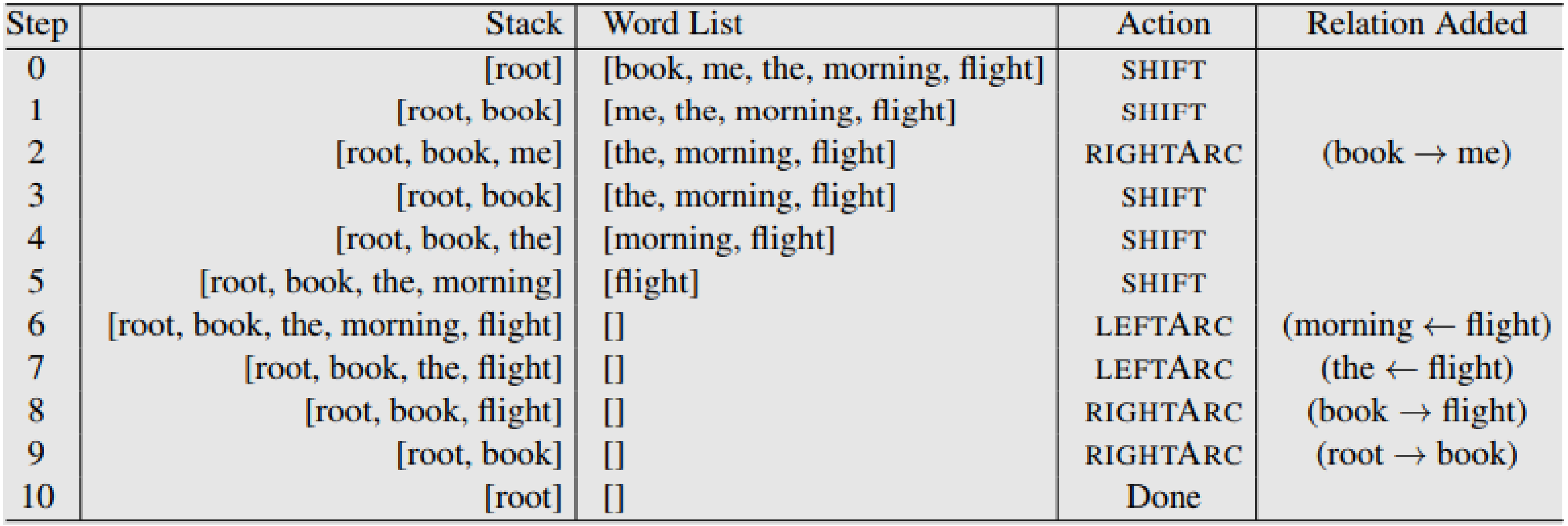
| Stack | Word List | Relations |
|---|---|---|
| [root, book, the, morning, flight] | [] | (book $\rightarrow$ me) |

Here, all the remaining words have been passed onto the stack and all that is left to do is to apply the appropriate reduce operators. In the current configuration, we employ the LEFTARC operator resulting in the following state.

| Stack | Word List | Relations |
|---|---|---|
| [root, book, the, flight] | [] | (book $\rightarrow$ me) |
| | | (morning $\leftarrow$ flight) |



# Trace of the Transition-based Parse



| Step | Stack | Word List | Action | Relation Added |
|---|---|---|---|---|
| 0 | [root] | [book, me, the, morning, flight] | SHIFT | |
| 1 | [root, book] | [me, the, morning, flight] | SHIFT | |
| 2 | [root, book, me] | [the, morning, flight] | RIGHTARC | (book $\rightarrow$ me) |
| 3 | [root, book] | [the, morning, flight] | SHIFT | |
| 4 | [root, book, the] | [morning, flight] | SHIFT | |
| 5 | [root, book, the, morning] | [flight] | SHIFT | |
| 6 | [root, book, the, morning, flight] | [] | LEFTARC | (morning $\leftarrow$ flight) |
| 7 | [root, book, the, flight] | [] | LEFTARC | (the $\leftarrow$ flight) |
| 8 | [root, book, flight] | [] | RIGHTARC | (book $\rightarrow$ flight) |
| 9 | [root, book] | [] | RIGHTARC | (root $\rightarrow$ book) |
| 10 | [root] | [] | Done | |

- for simplicity, we have illustrated this example without the labels on the dependency relations. To produce labeled trees, we can parameterize the LEFTARC and RIGHTARC operators with dependency labels, as in LEFTARC(NSUBJ) or RIGHTARC(DOBJ). This is equivalent to expanding the set of transition operators from our original set of three to a set that includes LEFTARC and RIGHTARC operators for each relation in the set of dependency relations being used, plus an additional one for the SHIFT operator. This, of course, makes the job of the oracle more difficult since it now has a much larger set of operators from which to choose.

## Creating an Oracle

- State-of-the-art transition-based systems use supervised machine learning methods to train classifiers that play the role of **the oracle**. Given appropriate training data, these methods learn a function that maps from configurations to transition operators.
  - But how can we train such classifiers?
  - we will need configurations paired with transition operators (i.e., LEFTARC, RIGHTARC, or SHIFT)
- Unfortunately, treebanks pair entire sentences with their corresponding trees, and therefore they don't directly provide what we need
  - Supply the oracle with: the training sentences to be parsed along with their corresponding reference parses from the treebank.
  - To produce training instances, we will then simulate the operation of the parser by running the algorithm and relying on **a new training oracle** to give us correct transition operators for each successive configuration

## How does the training oracle work?

Given:

- a reference parse and
- a configuration,

the training oracle proceeds as follows:

- Choose LEFTARC if it produces a correct head-dependent relation given the reference parse and the current configuration,
- Otherwise, choose RIGHTARC if :
  1. it produces a correct head-dependent relation given the reference parse and
  2. all of the dependents of the word at the top of the stack have already been assigned,
- Otherwise, choose SHIFT.

## Formalism

- During training the oracle has access to the following information:
  - A current configuration with a stack **S** and a set of dependency relations $\mathbf{R_c}$
  - A reference parse consisting of a set of vertices **V** and a set of dependency relations $\mathbf{R_p}$

Given this information, the oracle chooses transitions as follows:

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t.(S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

This allows us to deterministically record correct parser actions at each step as we progress through each training example, thereby creating the training set we require.

## Features

- By combining simple features, such as word forms or parts of speech, with specific locations in a configuration, we can employ the notion of a **feature template** that we've already encountered with part-of-speech tagging.

- Feature templates allow us to automatically generate large numbers of specific features from a training set. As an example, consider the following feature templates that are based on single positions in a configuration.

$$\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle$$
$$\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle$$

Individual features are denoted as **location.property**, where:

- s denotes the stack,
- b the word buffer, and
- r the set of relations.

Individual properties of locations include w for word forms, l for lemmas, and t for part-of-speech.

## Features

- Feature templates allow us to automatically generate large numbers of specific features from a training set. As an example, consider the following feature templates that are based on single positions in a configuration.

$$\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle$$
$$\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle$$

Individual features are denoted as **location.property**, where:

- s denotes the stack,
- b the word buffer, and
- r the set of relations.

Individual properties of locations include w for word forms, l for lemmas, and t for part-of-speech.

**For example**, the feature corresponding to the word form at the top of the stack would be denoted as s1.w, and the part of speech tag at the front of the buffer b1.t. We can also combine individual features via concatenation into more specific features that may prove useful. For example, the feature designated by s1.wt represents the word form concatenated with the part of speech of the word at the top of the stack. Finally, op stands for the transition operator for the training example in question (i.e., the label for the training instance).

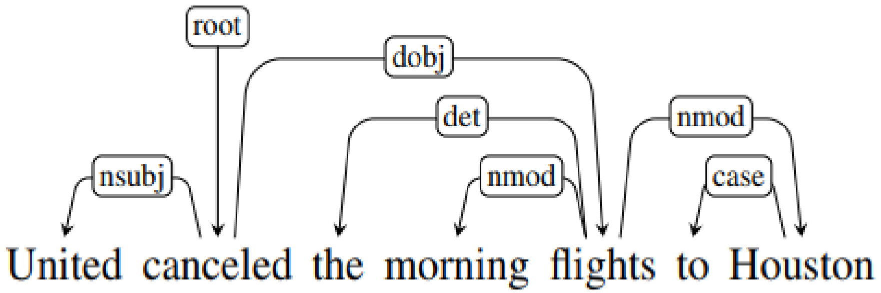
## Example

We can also combine individual features via concatenation into more specific features that may prove useful.

For example, the feature designated by s1.wt represents the word form concatenated with the part of speech of the word at the top of the stack. Finally, op stands for the transition operator for the training example in question (i.e., the label for the training instance).

$$\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle$$
$$\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle$$

Let's consider the simple set of single-element feature templates given above in the context of the following intermediate configuration derived from a training oracle for example



| Stack | Word buffer | Relations |
|---|---|---|
| [root, canceled, flights] | [to Houston] | (canceled → United) |
| | | (flights → morning) |
| | | (flights → the) |

Operator???
**SHIFT**

## The Application of the Feature Template

The application of our set of feature templates to this configuration would result in the following set of instantiated features.

$$\begin{aligned}
\langle s_1.w &= \mathit{flights}, op = \mathit{shift}\rangle \\
\langle s_2.w &= \mathit{canceled}, op = \mathit{shift}\rangle \\
\langle s_1.t &= \mathit{NNS}, op = \mathit{shift}\rangle \\
\langle s_2.t &= \mathit{VBD}, op = \mathit{shift}\rangle \\
\langle b_1.w &= \mathit{to}, op = \mathit{shift}\rangle \\
\langle b_1.t &= \mathit{TO}, op = \mathit{shift}\rangle \\
\langle s_1.wt &= \mathit{flightsNNS}, op = \mathit{shift}\rangle
\end{aligned}$$

| Stack | Word buffer | Relations |
|---|---|---|
| [root, canceled, flights] | [to Houston] | (canceled $\rightarrow$ United) |
| | | (flights $\rightarrow$ morning) |
| | | (flights $\rightarrow$ the) |

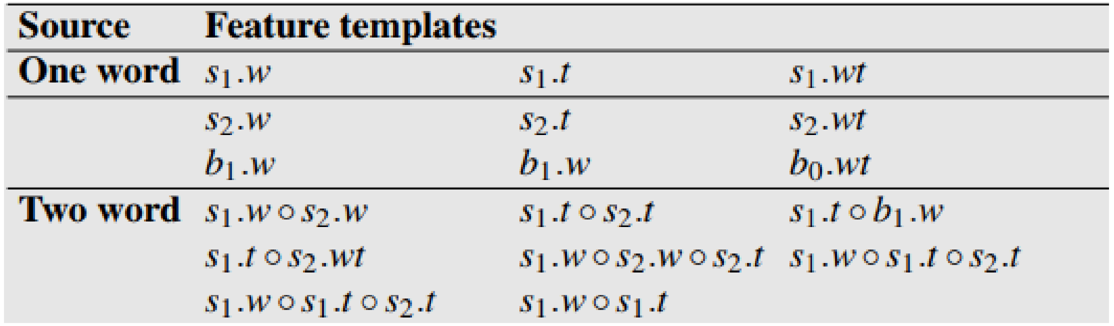
Given that the left and right arc transitions operate on the top two elements of the stack, features that combine properties from these positions are even more useful. For example, a feature like s1.t ◦ s2.t concatenates the part of speech tag of the word at the top of the stack with the tag of the word beneath it.

$$\langle s_1.t \circ s_2.t = \mathit{NNSVBD}, op = \mathit{shift}\rangle$$

# Dynamic Features

Some of the dependency parsing features make use of dynamic features — features such as head words and dependency relations that have been predicted at earlier steps in the parsing process, as opposed to features that are derived from static properties of the input.

Standard feature templates for training transition-based dependency parsers:



| **Source** | **Feature templates** | | |
|---|---|---|---|
| **One word** | $s_1.w$ | $s_1.t$ | $s_1.wt$ |
| | $s_2.w$ | $s_2.t$ | $s_2.wt$ |
| | $b_1.w$ | $b_1.w$ | $b_0.wt$ |
| **Two word** | $s_1.w \circ s_2.w$ | $s_1.t \circ s_2.t$ | $s_1.t \circ b_1.w$ |
| | $s_1.t \circ s_2.wt$ | $s_1.w \circ s_2.w \circ s_2.t$ | $s_1.w \circ s_1.t \circ s_2.t$ |
| | $s_1.w \circ s_1.t \circ s_2.t$ | $s_1.w \circ s_1.t$ | |

**Learning**: the dominant approaches to training transition-based dependency parsers have been multinomial logistic regression and support vector machines, both of which can make effective use of large numbers of sparse features.

More recently, neural networks have been applied successfully to transition-based parsing (Chen and Manning, 2014). These approaches eliminate the need for complex, hand-crafted features and have been particularly effective at overcoming the data sparsity issues normally associated with training transition-based parsers.

## Summary

This chapter has introduced the concept of **dependency grammars** and **dependency parsing.**

Here’s a summary of the main points that we covered:

- In dependency-based approaches to syntax, the structure of a sentence is described in terms of a set of binary relations that hold between the words in a sentence. Larger notions of constituency are not directly encoded in dependency analyses.
- The relations in a dependency structure capture the head-dependent relationship among the words in a sentence.
- Dependency-based analyses provides information directly useful in further language processing tasks including information extraction, semantic parsing and question answering
- Transition-based parsing systems employ a greedy stack-based algorithm to create dependency structures