Dependency Parsing and Logical Representations of Sentence Meaning

Natalie Parde

UIC CS 421

What is dependency parsing?

- Automatically determining **directed grammatical and semantic relationships** between words
  - **Semantic:** Focused on **meaning**
- This information is useful for many NLP applications, including:
  - Coreference resolution
  - Question answering
  - Information extraction

How are dependency grammars different from CFGs?

- CFGs generate constituent-based representations
  - Noun phrases, verb phrases, etc.
  - These tell us about the syntactic structure, rather than the semantic relationship between words
- Dependency grammars define sentence structure in terms of the relationships between individual words
  - Nominal subject, direct object, etc.
- For both, labels are still drawn from a fixed inventory of grammatical relations

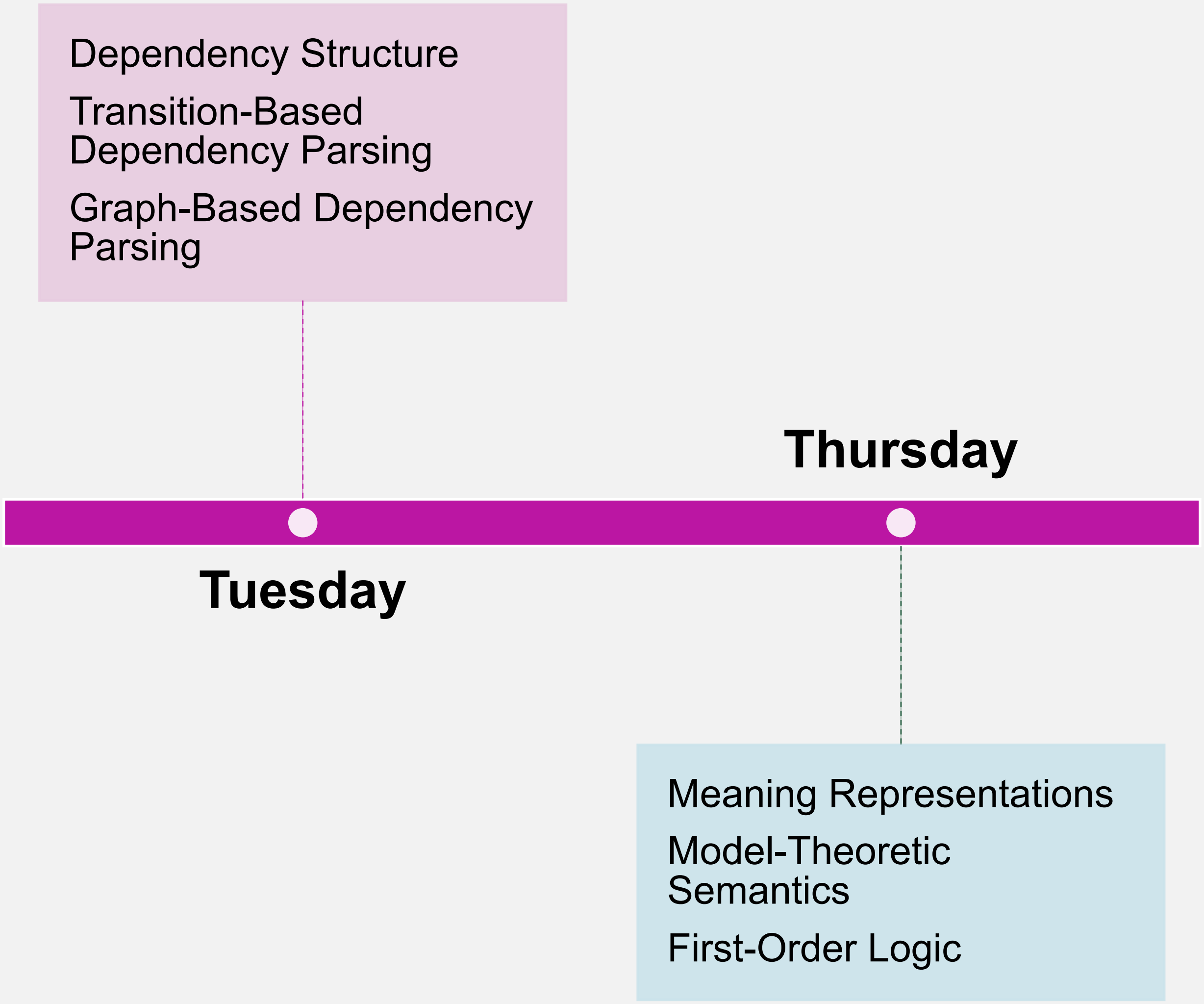
Dependency grammars are especially helpful for interpreting morphologically rich languages with a relatively free word order.

Morphologically rich: Grammatical relationships are indicated by changes to words, rather than sentence position

Free word order: Words can be moved around in a sentence but the overall meaning will remain the same (less reliance on syntax)

Typically, languages that are morphologically richer have less strict syntactic rules

This Week's Topics



Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Tuesday

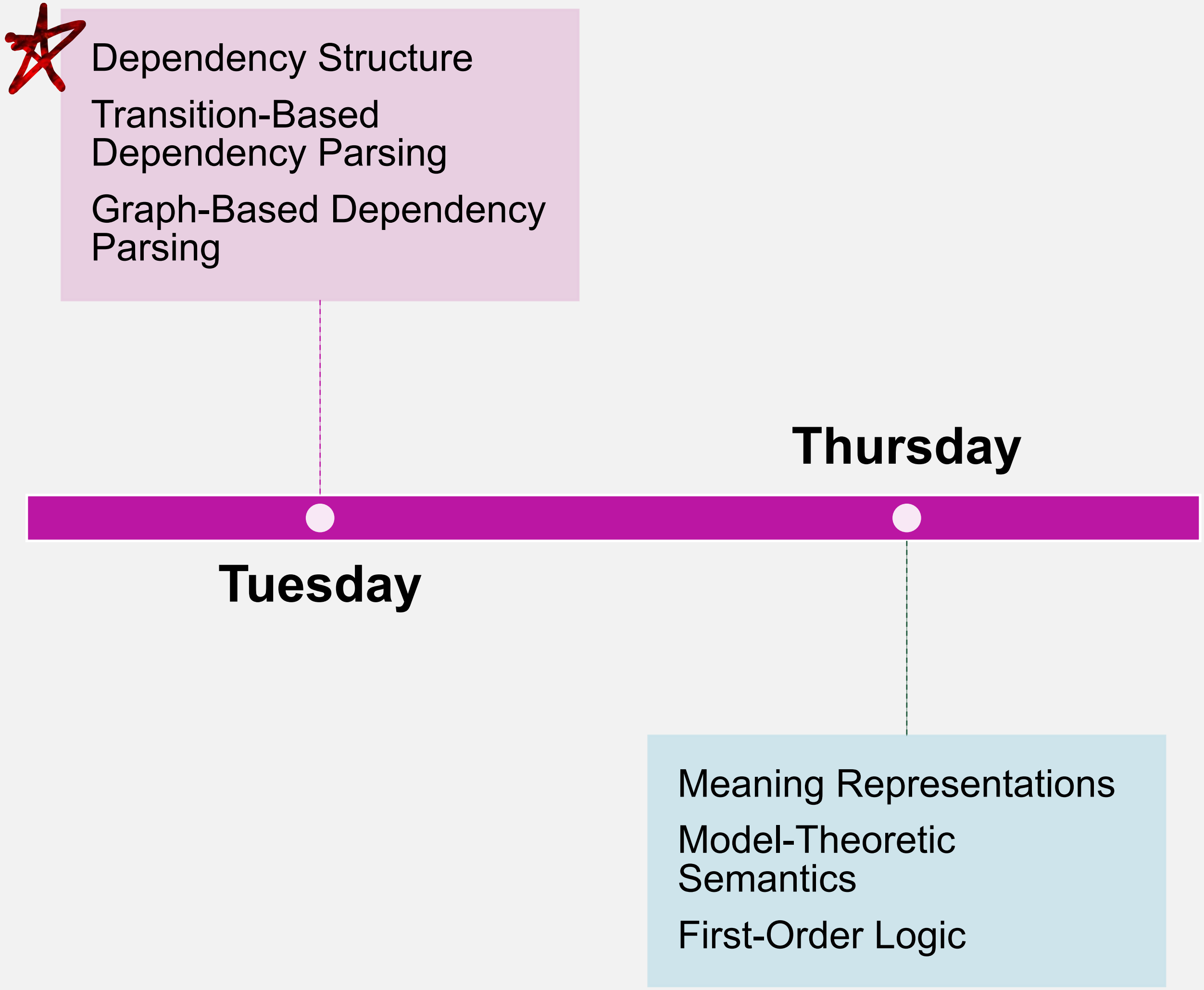
Thursday

Meaning Representations

Model-Theoretic Semantics

First-Order Logic

This Week's Topics



Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Tuesday

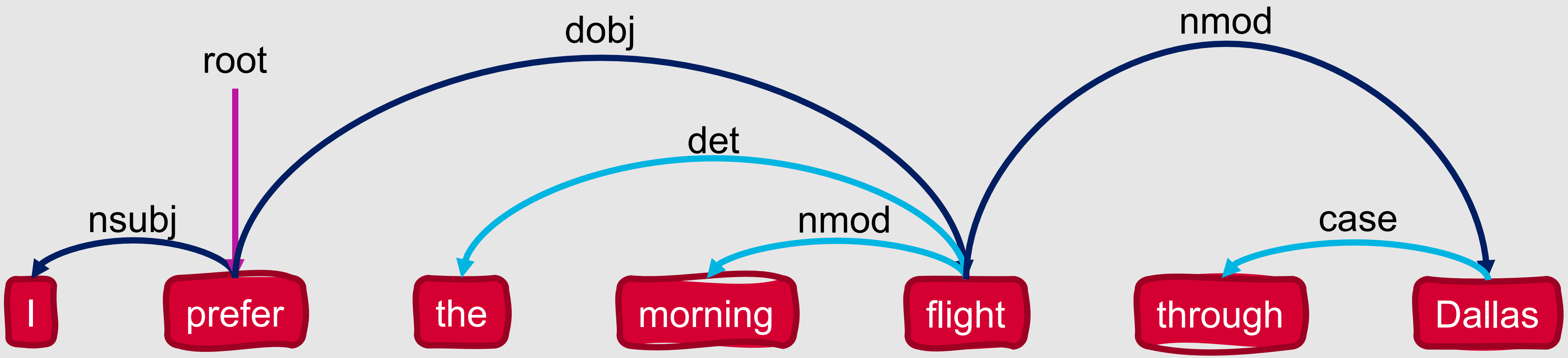
Thursday

Meaning Representations

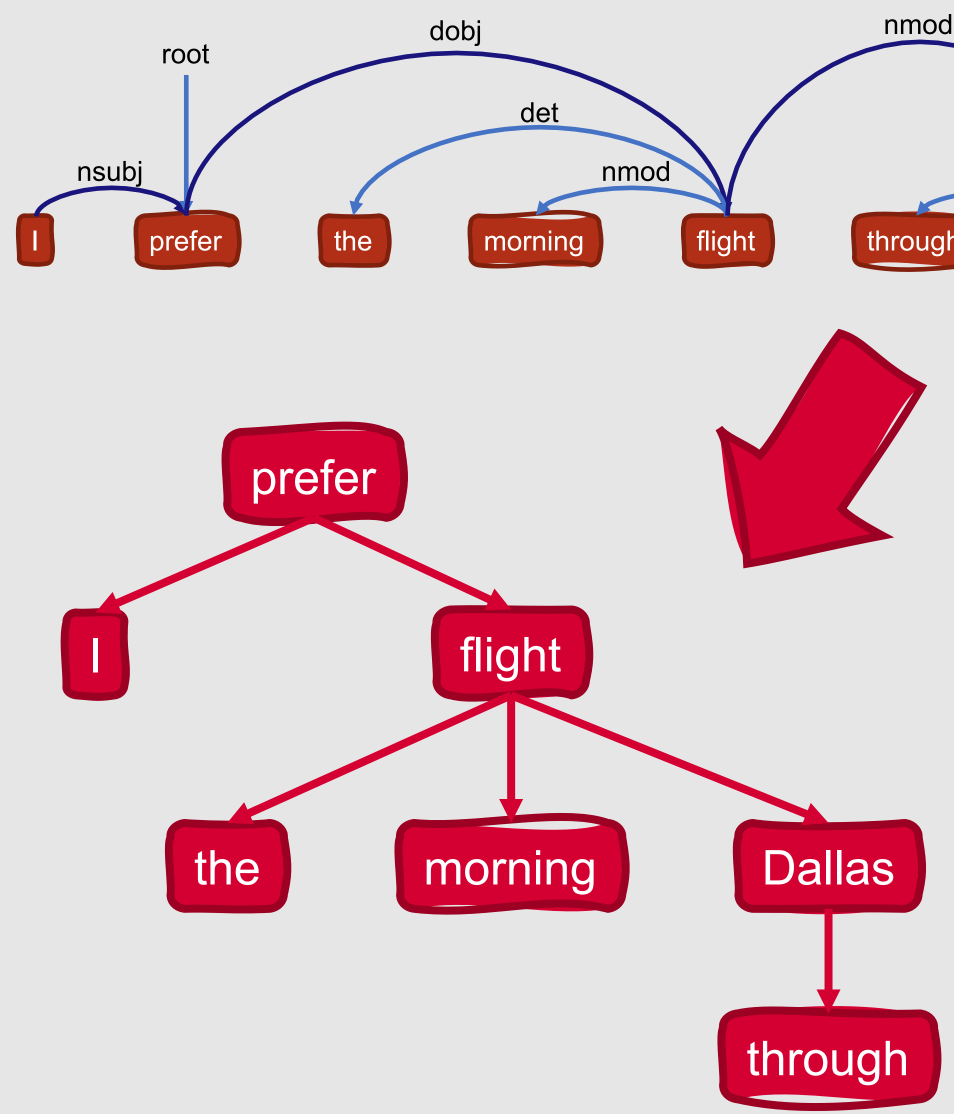
Model-Theoretic Semantics

First-Order Logic

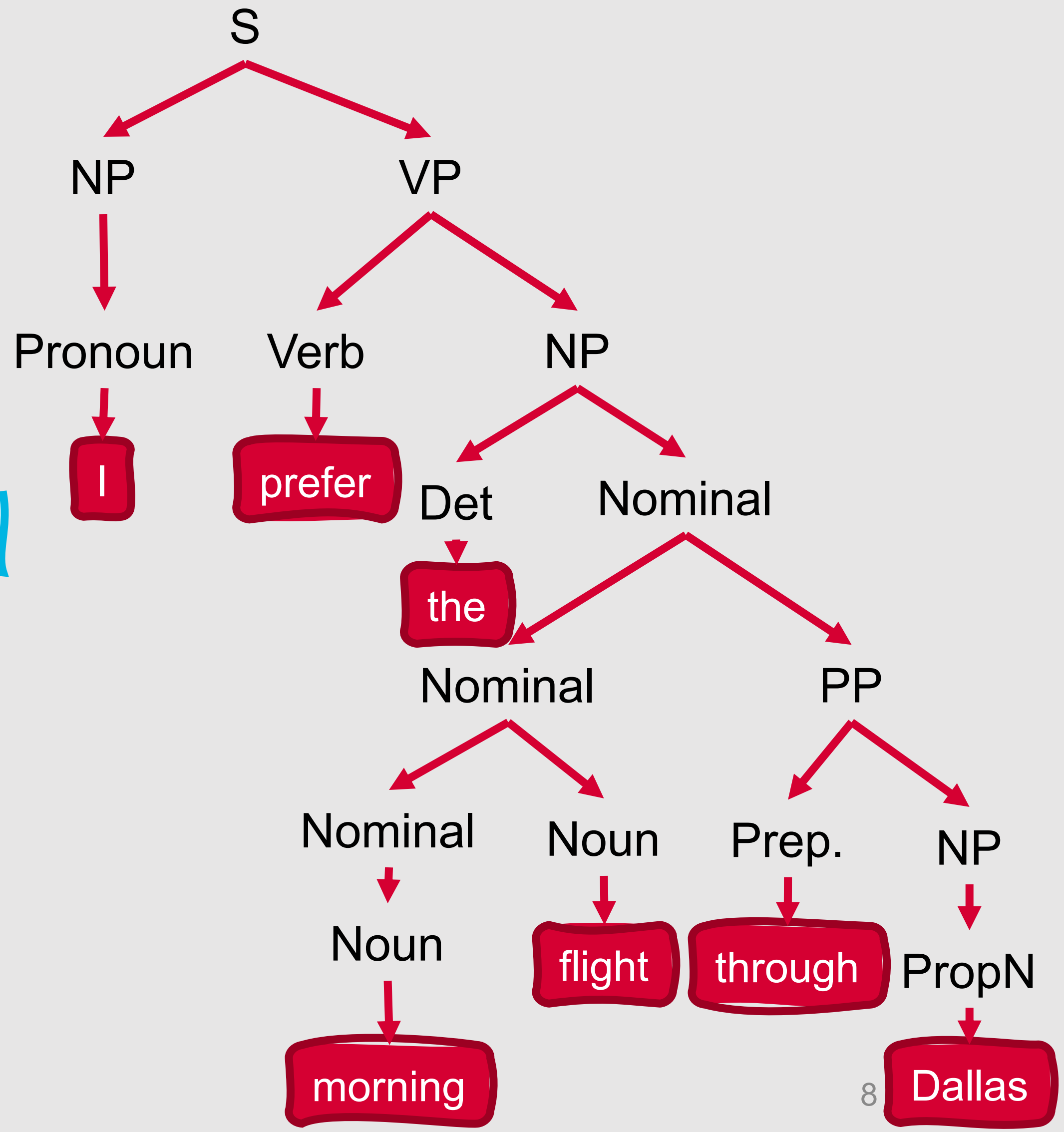
Typed Dependency Structure



Comparison with Syntactic Parse



vs.



Dependency Relations

- Two components:
  - **Head**
  - **Dependent**
- **Heads** are linked to the words that are immediately **dependent** on them
- Relation types describe the **dependent**'s role with respect to its **head**
  - Subject
  - Direct object
  - Indirect object

Dependency Relations

- Relation types *tend* to correlate with sentence position and constituent type in English, but there is not an explicit connection between these elements
- In languages with relatively free word order, the information encoded in these relation types often cannot be estimated from constituency trees

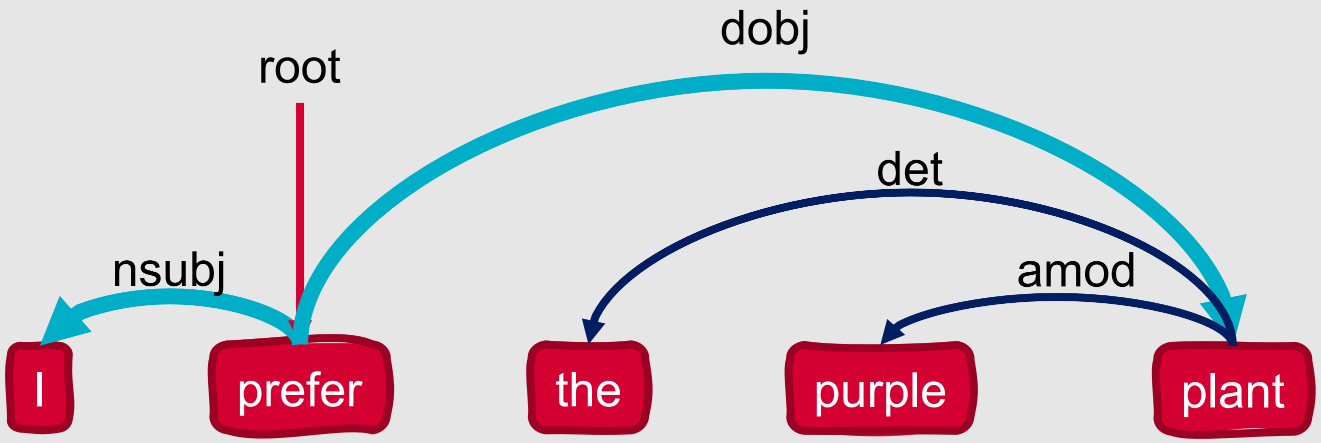
Just like with CFGs, there are a variety of taxonomies that can be used to label dependencies between words.

- A couple of the most popular **dependency treebanks and tagsets** include:
  - Stanford dependencies
    - https://downloads.cs.stanford.edu/nlp/software/dependencies_manual.pdf
  - Universal dependencies
    - https://universaldependencies.org/

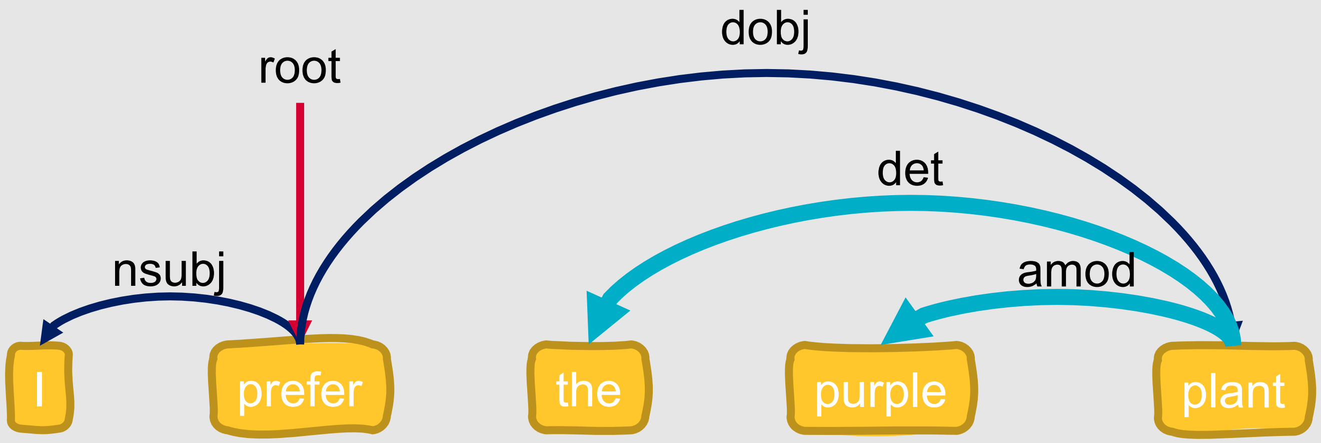
Recently, most researchers have moved toward using universal dependencies.

- Universal dependencies can be broken into:
  - **Clausal Relations:** Describe syntactic roles that say something about the predicate
  - **Modifier Relations:** Describe the ways that words can modify their heads

Clausal Relations



Modifier Relations



Universal Dependencies

	Structural categories of dependent			
Functional categories w.r.t. head	**Nominals**	**Clauses**	**Modifier Words**	**Function Words**
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated	advcl	advmod discourse	aux cop mark
Dependents of Nominals	nmod appos nummod	acl	amod	det clf case

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated		od rse	aux cop mark
Dependents of Nominals	nmod appos nummod			det clf case

Natalie wrote a dissertation.
nsubj(wrote, Natalie)

Natalie wrote a dissertation.
obj(wrote, dissertation)

Natalie wrote UIC a dissertation.
iobj(wrote, UIC)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated		mod ourse	aux cop mark
Dependents of Nominals	nmod appos nummod			et clf case

Natalie wrote a dissertation for UIC.
obl(wrote, UIC)

UIC, read my dissertation!
vocative(read, UIC)

There is nothing but praise for the dissertation.
expl(nothing, there)

You must not eat it, the dissertation.
dislocated(eat, dissertation)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj			
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated		e	aux cop mark
Dependents of Nominals	nmod appos nummod		d	det clf case

The purpose of this dissertation is to determine the best homework strategy.
nmod(purpose, dissertation)

My school, UIC, is in Chicago.
appos(school, UIC)

UIC has 34,000 students.
nummod(students, 34,000)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated	advcl	[illegible]	aux
Dependents of Nominals	nmod appos nummod	acl		det e

What she said about starting the project makes sense.
csubj(makes, said)

She said you should start it now.
ccomp(said, start)

I consider it already done.
xcomp(consider, done)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated	advcl	advmod discourse	aux cop mark
Dependents of Nominals	nmod appos nummod	acl	amod	det clf case

He was upset when she read her dissertation to him.
advcl(upset, read)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Functional categories w.r.t. head

Structural categories of dependent

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated	advcl		mark
Dependents of Nominals	nmod appos nummod	acl	amod	det clf case

There is a document discussing the assignment.
acl(document, discussing)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates				
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated	advcl	advmod discourse	aux cop mark
Dependents of Nominals			amod	det clf case

UIC quickly emailed the students about the day off.
advmod(emailed, quickly)

She said, “Well, let’s schedule a meeting.”
discourse(schedule, well)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details):
conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	expl dislocated	advcl	advmod discourse	aux cop mark
Dependents of Nominals	nmod appos nummod	acl	amod	det clf case

He read the extensive syllabus.
amod(syllabus, extensive)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj	csubj		
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated			aux cop mark
Dependents of Nominals	nummod			det clf case

UIC had closed the campus for the break.
aux(closed, had)

It was good to have some time off.
cop(good, was)

They knew that this would refresh everyone for the spring.
mark(refresh, that)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Universal Dependencies

Structural categories of dependent

Functional categories w.r.t. head

	Nominals	Clauses	Modifier Words	Function Words
Core Arguments of Clausal Predicates	nsubj obj iobj	csubj ccomp xcomp		
Non-Core Dependents of Clausal Predicates	obl vo dis		dvmod scourse	aux cop mark
Dependents of Nominals	nmod appos nummo	acl	amod	det clf case

That was the goal.
det(goal, the)

A word that accompanies a noun to reflect some conceptual classification of the noun (not used in English)

Everyone went on vacation after that.
case(that, after)

Other miscellaneous dependency relations (see https://universaldependencies.org/u/dep/index.html for details): conj, cc, fixed, flat, compound, list, parataxis, orphan, goeswith, reparandum, punct, root, dep

Dependency Formalisms

Dependency structures are directed graphs

- $G = (V, A)$
  - V is a set of vertices
  - A is a set of ordered pairs of vertices, or arcs
- V corresponds to the words in a sentence
  - May also include punctuation
  - In morphologically rich languages, may include stems and affixes
- Arcs capture the grammatical relationships between those words

In general, dependency structures:

- Must be connected
- Must have a designated root node
- Must be acyclic

Dependency Trees

- Directed graphs (such as those we've seen already) that satisfy the following constraints:
    - Single designated root node
        - No incoming arcs to the root!
    - All vertices *except the root node* have exactly one incoming arc
    - There is a unique path from the root node to each vertex

How to translate from constituent to dependency structures?

Two steps:

1. Identify all head-dependent relations in the constituent tree
2. Identify the correct dependency relations for those head-dependent pairs

This can by done by:

- Marking the **head child** of each node in a phrase structure, based on a set of rules
- In the dependency structure, make the head of each **non-head child** depend on the head of the **head child**

- However, doing this can produce results that are far from perfect!
  - Most noun phrases do not have much (or any) internal structure
  - Morphological information generally isn’t encoded in phrase structure trees

Types of Dependency Parsers

Transition

Transition-based

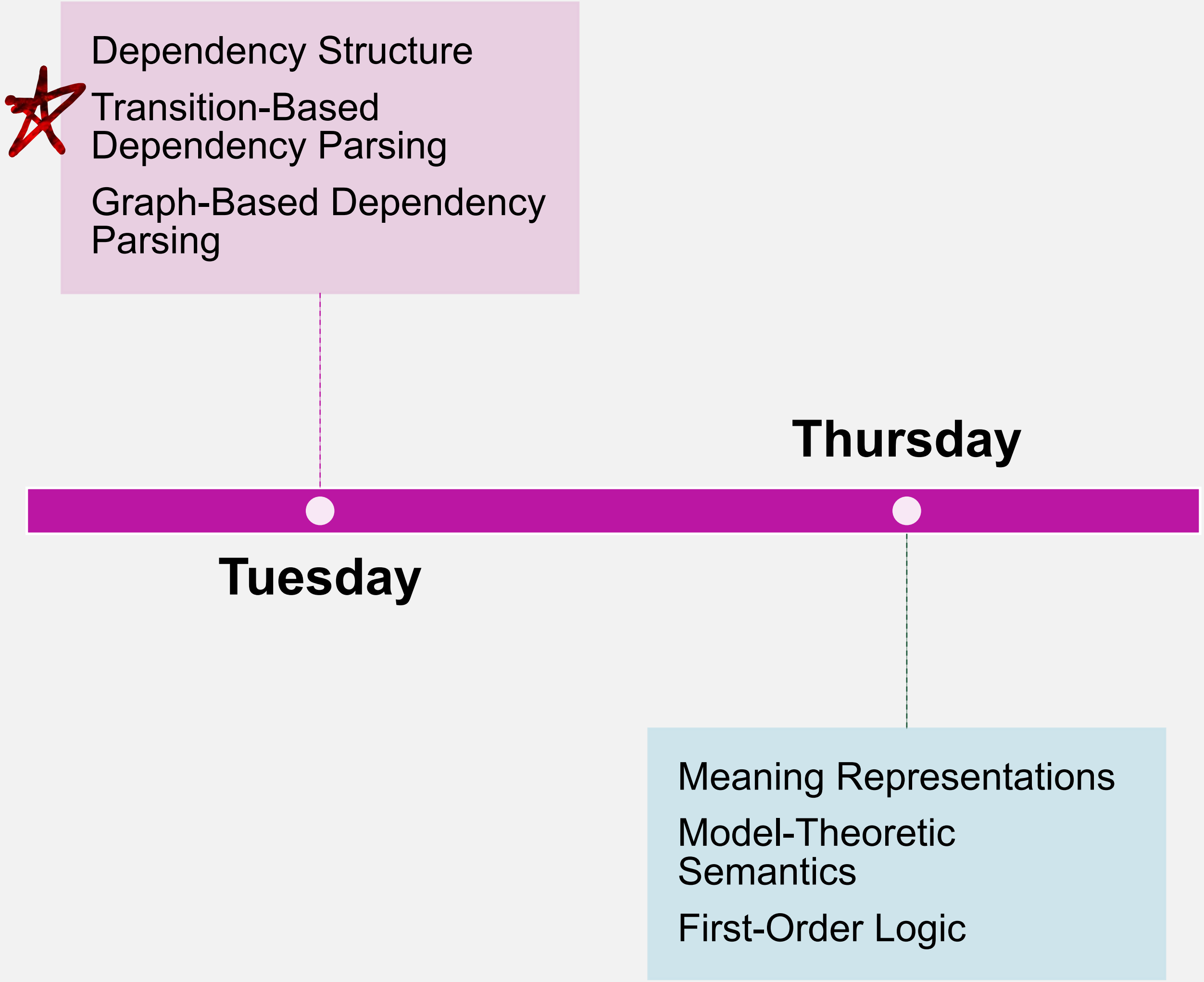
- Build a single tree in a beginning-to-end sweep over the input sentence

Graph

Graph-based

- Search through the space of possible trees for a given sentence, and try to find the tree that maximizes some score

This Week's Topics



Tuesday

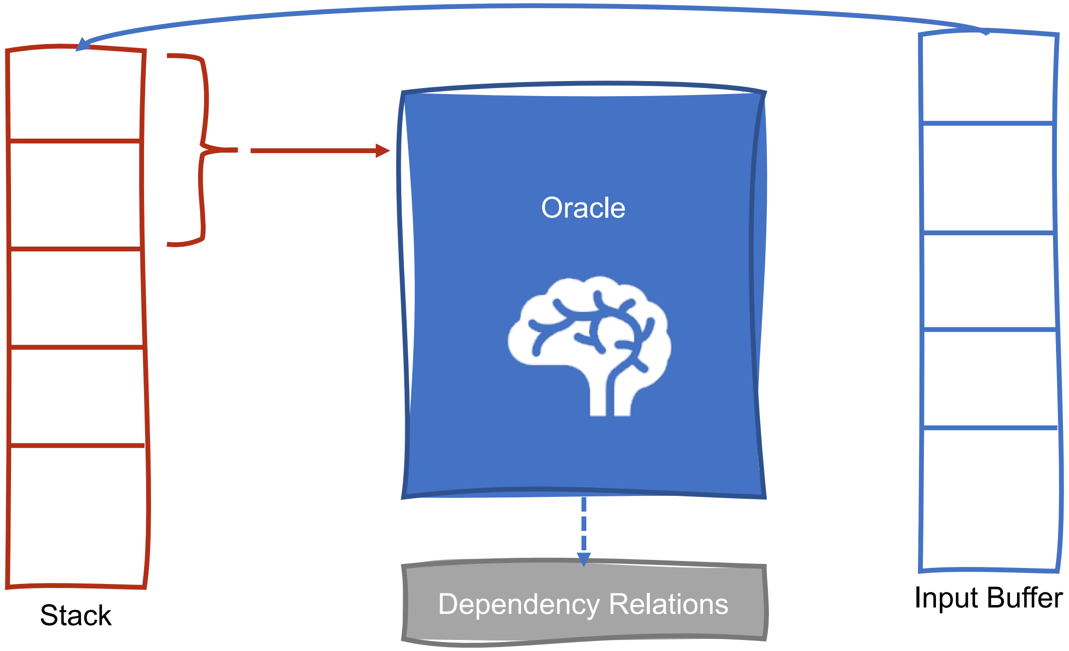
- Dependency Structure
- Transition-Based Dependency Parsing
- Graph-Based Dependency Parsing

Thursday

- Meaning Representations
- Model-Theoretic Semantics
- First-Order Logic

Transition-based Dependency Parsing

- Earliest transition-based approach: **shift-reduce parsing**
    - Input tokens are successively shifted onto a stack
    - The two top elements of the stack are matched against a set of possible relations provided by some knowledge source
    - When a match is found, a head-dependent relation between the matched elements is asserted
- Goal is to find a final parse that accounts for all words



Transition-based Parsing

- We can build upon shift-reduce parsing by defining **transition operators** to guide the parser’s decisions
- Transition operators work by producing new **configurations**:
  - Stack
  - Input buffer of words
  - Set of relations representing a dependency tree

Transition-based Parsing

Initial configuration:

- Stack contains the ROOT node
- Input buffer is initialized with all words in the sentence, in order
- Empty set of relations represents the parse

Final configuration:

- Stack should be empty (except ROOT)
- Input buffer should be empty
- Set of relations represents the parse

Operators

- The operators used in transition-based parsing then perform one of the following tasks:
  - **Assign the current word as the head of some other word** that has already been seen
  - **Assign some other word that has already been seen as the head** of the current word
  - **Do nothing** with the current word

Operators

- More formally, these operators are defined as:
  - **LeftArc:** Asserts a head-dependent relation between the word at the top of the stack and the word directly beneath it (the second word), and removes the second word from the stack
    - Cannot be applied when ROOT is the second element in the stack
    - Requires two elements on the stack
  - **RightArc:** Asserts a head-dependent relation between the second word and the word at the top of the stack, and removes the word at the top of the stack
    - Requires two elements on the stack
  - **Shift:** Removes a word from the front of the input buffer and pushes it onto the stack
- These operators implement the **arc standard approach** to transition-based parsing

Arc Standard Approach to Transition-based Parsing

- Notable characteristics:
  - Transition operators only assert relations between elements at the top of the stack
  - Once an element has been assigned its head, it is removed from the stack
    - Not available for further processing!
- Benefits:
  - Reasonably effective
  - Simple to implement

Formal Algorithm: Arc Standard Approach

```
state ← {[root], [words], []}
while state not final:
     # Choose which transition operator to apply
    transition ← oracle(state)

     # Apply the operator and create a new state
    state ← apply(transition, state)
```

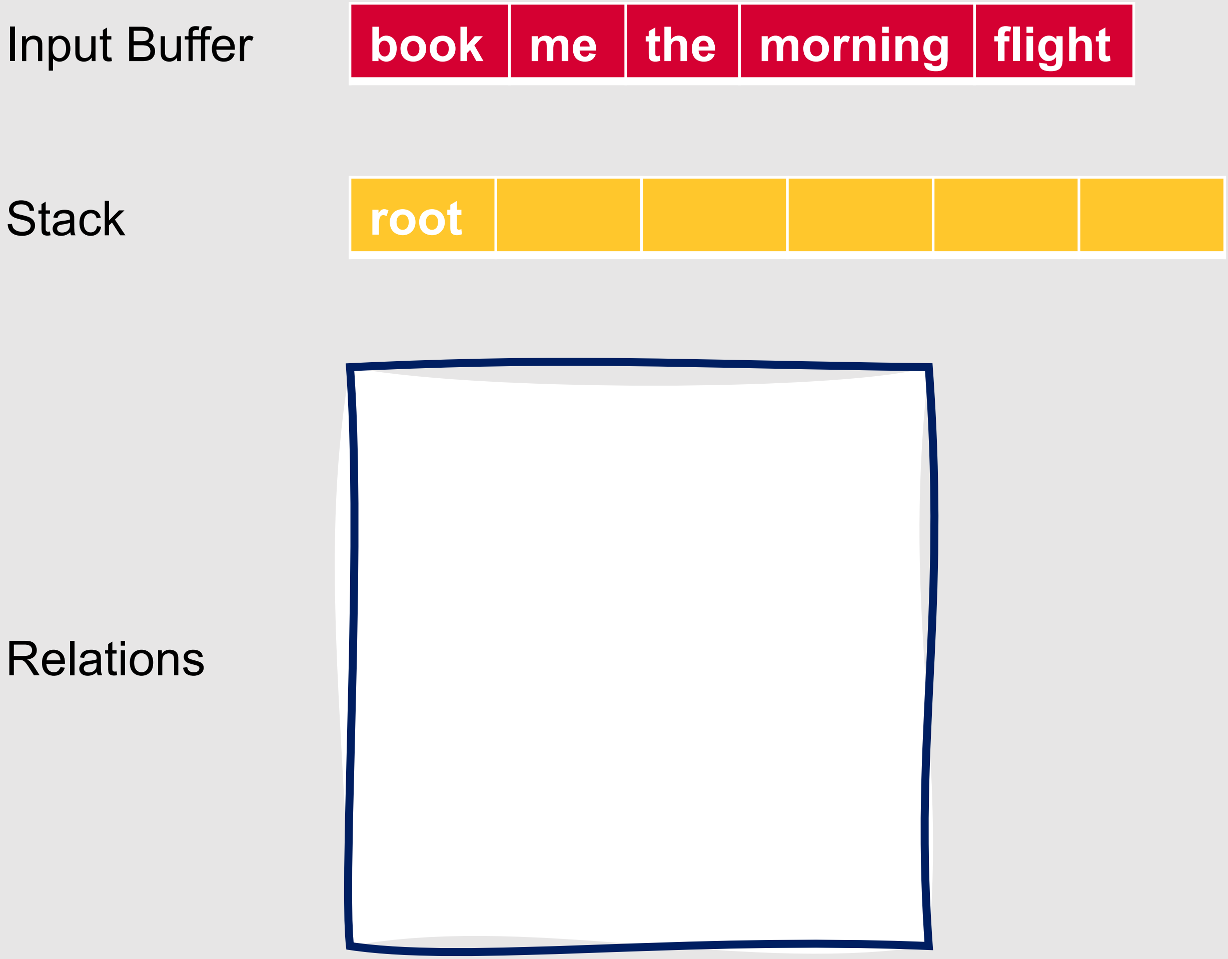
Process ends when:

- All words in the sentence have been consumed
- The ROOT node is the only element remaining on the stack

This is *not* an example of dynamic programming!

- The arc standard approach is a **greedy algorithm**
  - Oracle chooses a single operation at each step
  - Parser proceeds with that choice
    - No other options explored
    - No backtracking
  - Single parse returned at the end

Arc Standard: Example



Input Buffer

book	me	the	morning	flight

Stack

root					

Relations

Arc Standard: Example

Input Buffer		me	the	morning	flight

Stack	book	root				

Relations

Only one item in the stack!

Shift **book** from the input buffer to the stack

Arc Standard: Example

Input Buffer

		the	morning	flight

Stack

me	book	root			

Relations

Valid options: Shift, RightArc

Oracle selects Shift

Shift **me** from the input buffer to the stack

Arc Standard: Example

Input Buffer

		the	**morning**	**flight**

Stack

book	**root**				

Relations

(book → me)

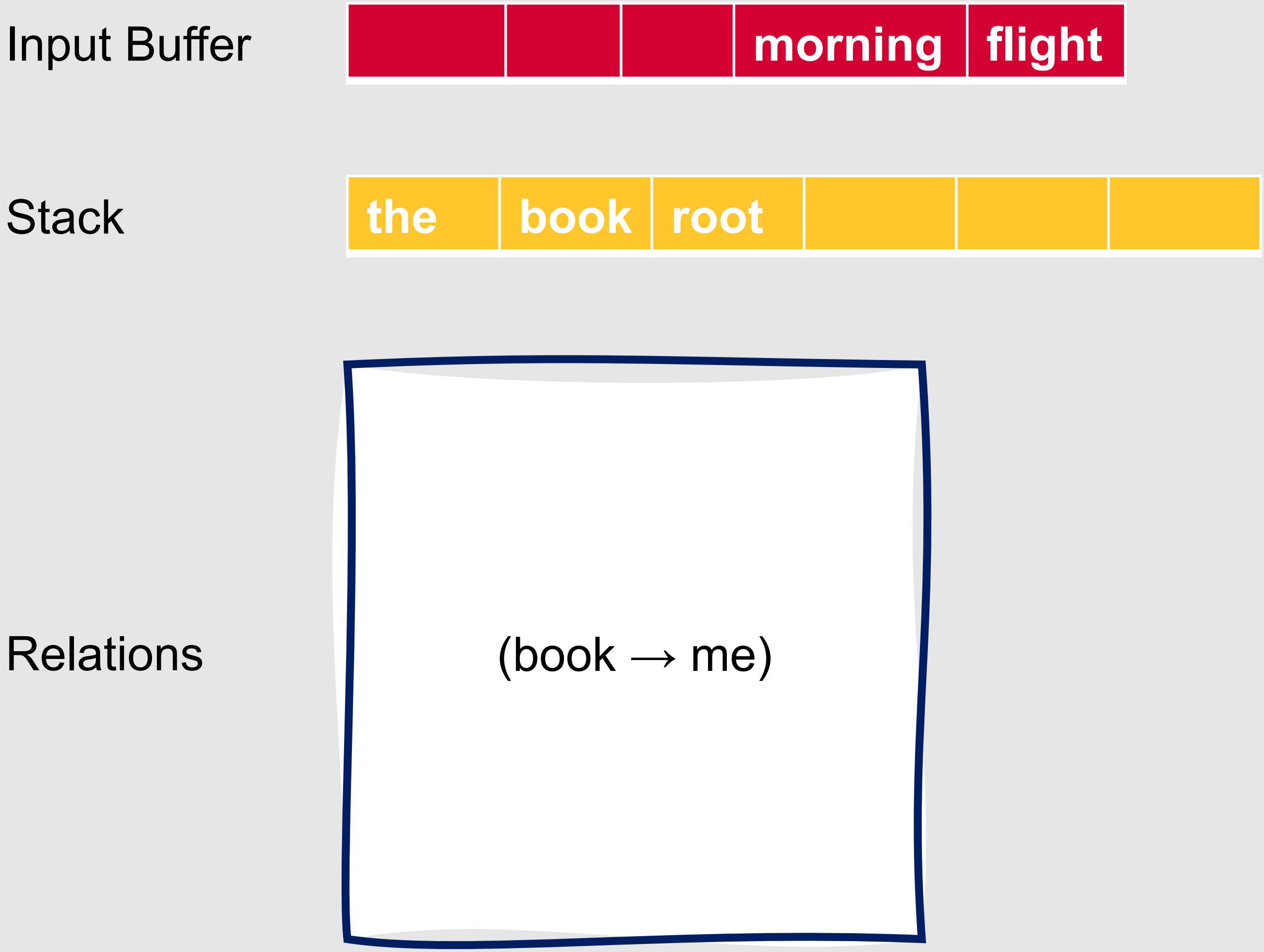
Valid options: Shift, RightArc, LeftArc

Oracle selects RightArc

Remove **me** from the stack

Add relation (book → me) to the set of relations

Arc Standard: Example



Input Buffer: morning | flight

Stack: the | book | root

Relations: (book → me)

Valid options: Shift, RightArc

Oracle selects Shift

Shift **the** from the input buffer to the stack

Arc Standard: Example

Input Buffer

				flight

Stack

morning	**the**	**book**	**root**		

Relations

(book → me)

Valid options: Shift, RightArc, LeftArc

Oracle selects Shift

Shift **morning** from the input buffer to the stack

Arc Standard: Example

Input Buffer

Stack

flight	morning	the	book	root	

Relations

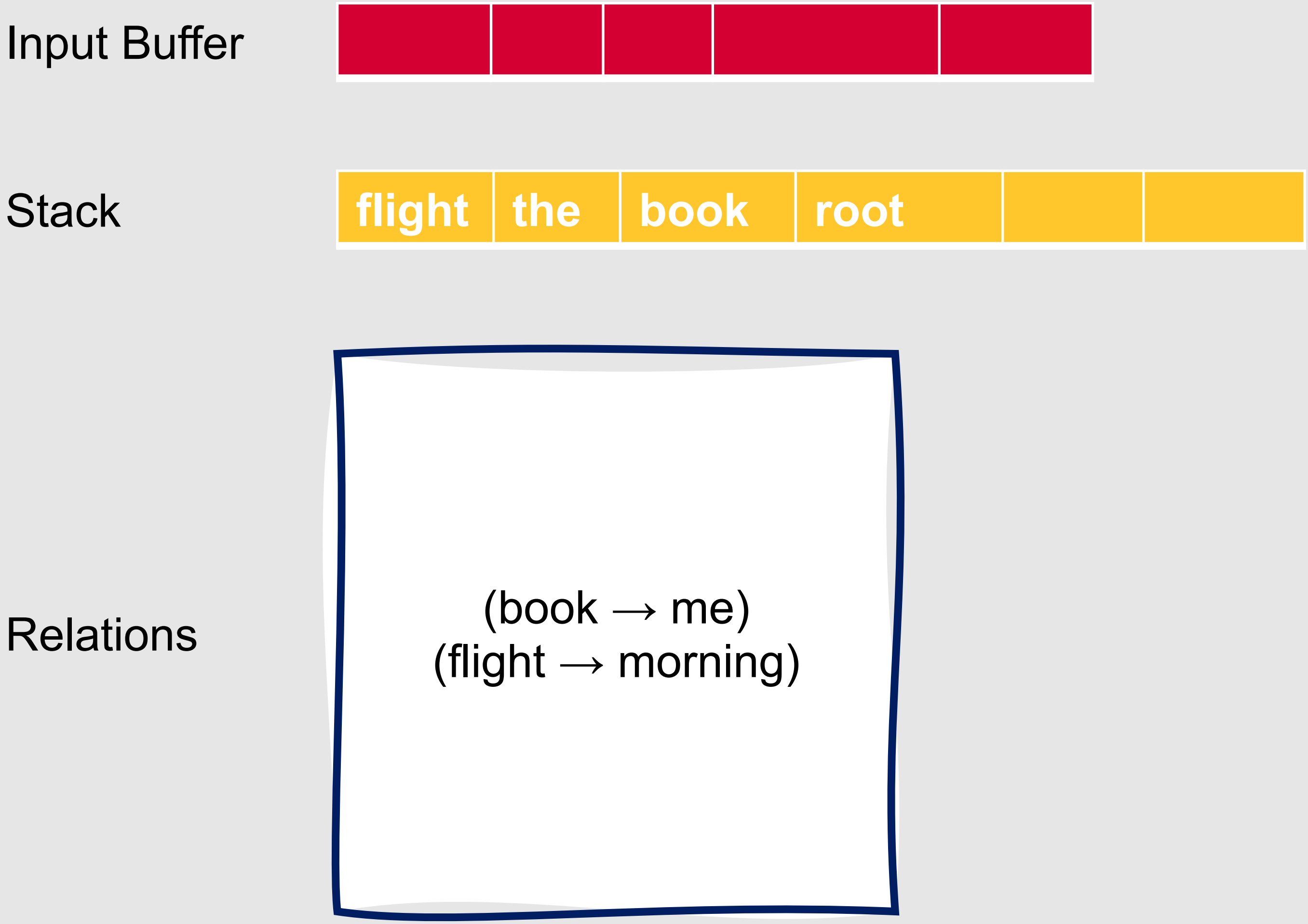
(book → me)

Valid options: Shift, RightArc, LeftArc

Oracle selects Shift

Shift **flight** from the input buffer to the stack

Arc Standard: Example



Input Buffer

Stack

flight	the	book	root		

Relations

(book → me)
(flight → morning)

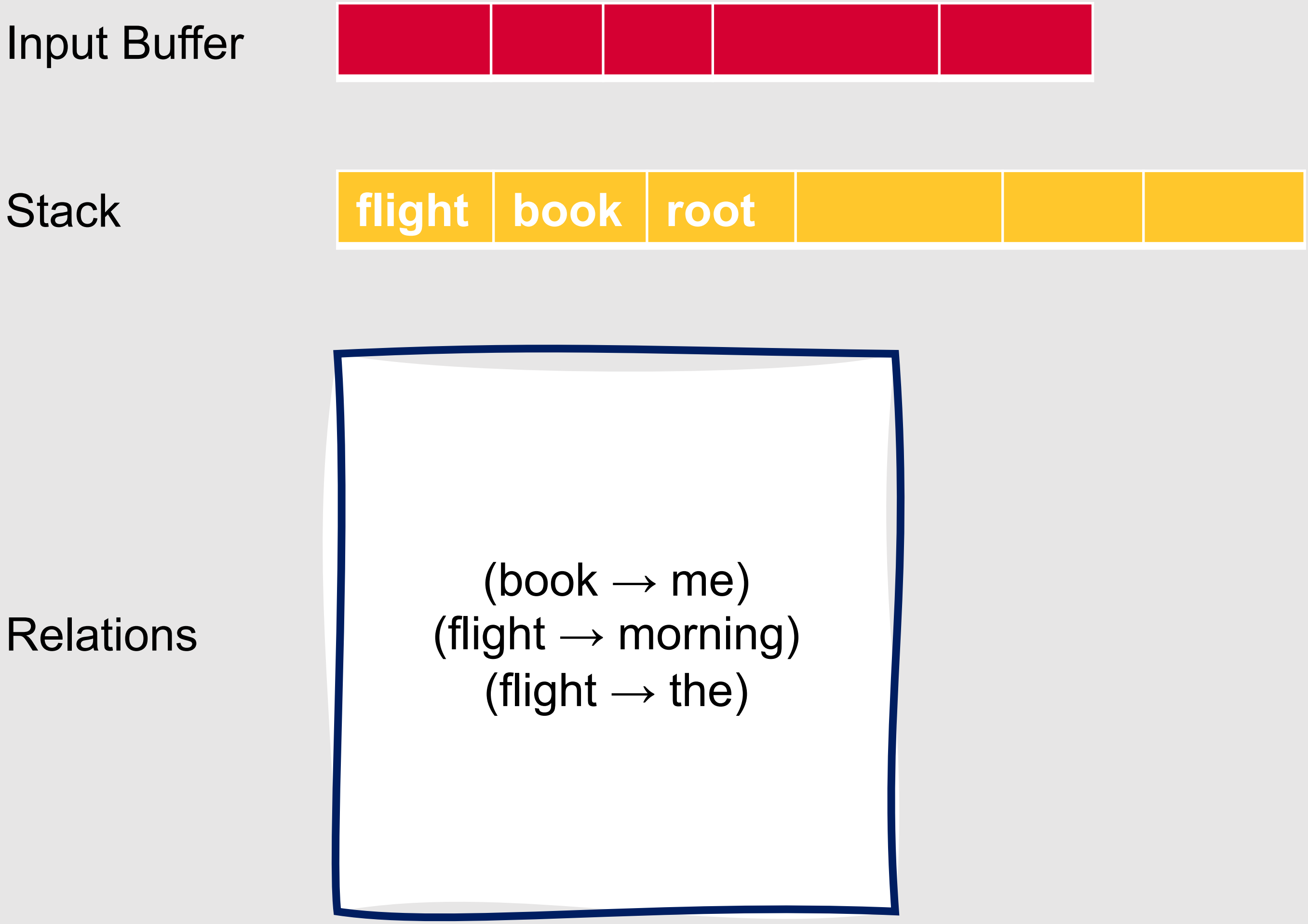
Valid options: RightArc, LeftArc

Oracle selects LeftArc

Remove **morning** from the stack

Add relation (flight → morning) to the set of relations

Arc Standard: Example



Input Buffer

Stack

flight	book	root			

Relations

(book → me)
(flight → morning)
(flight → the)

Valid options: RightArc, LeftArc

Oracle selects LeftArc

Remove **the** from the stack

Add relation (flight → the) to the set of relations

Arc Standard: Example

Input Buffer						
Stack	book	root				

Relations

(book → me)
(flight → morning)
(flight → the)
(book → flight)

Valid options: RightArc, LeftArc

Oracle selects RightArc

Remove **flight** from the stack

Add relation (book → flight) to the set of relations

Arc Standard: Example

Input Buffer

Stack

root

Relations

(book → me)
(flight → morning)
(flight → the)
(book → flight)
(root → book)

Valid options: RightArc

Oracle selects RightArc

Remove **book** from the stack

Add relation (root → book) to the set of relations

Arc Standard: Example

Input Buffer

Stack

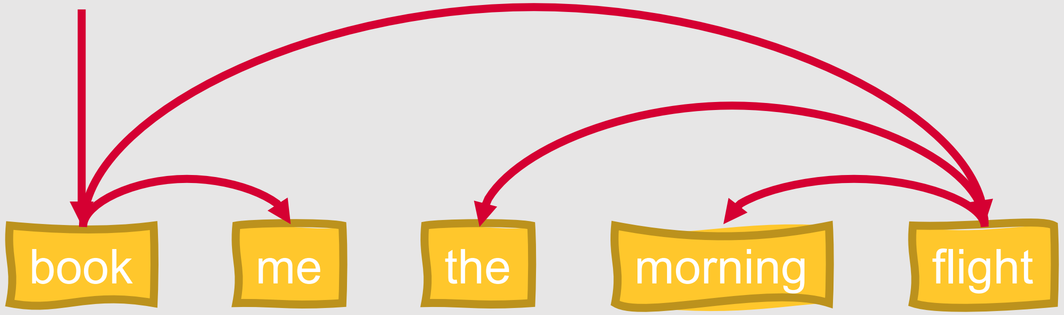
root

Valid options: None

State is final

Relations

(book → me)
(flight → morning)
(flight → the)
(book → flight)
(root → book)



A few things worth noting….

- We assumed in the previous example that our oracle was always correct …this is not necessarily (or perhaps not even likely) the case!
  - Incorrect choices lead to incorrect parses since the algorithm cannot perform any backtracking
- Alternate sequences may also lead to equally valid parses

How do we get actual dependency labels?

- Parameterize **LeftArc** and **RightArc**
  - LeftArc(nsubj), RightArc(obj), etc.
- Of course, this makes the oracle’s job more difficult (much larger set of operators from which to choose!)

iobj(book → me)
compound(flight → morning)
det(flight → the)
obj(book → flight)
root(root → book)

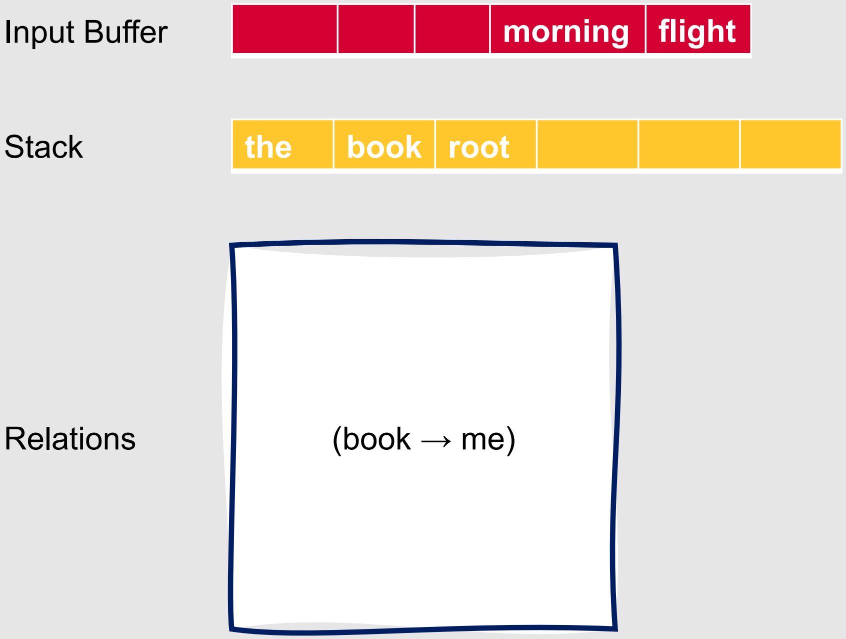
How does the oracle know what to choose?

- Generally, systems use **supervised machine learning** for this task
- This requires a training set of configurations labeled with correct transition operators
- The oracle learns which transitions to predict for previously-unseen configurations based on extracted features and/or representations for labeled configurations in the training set

What types of machine learning models are used as oracles?

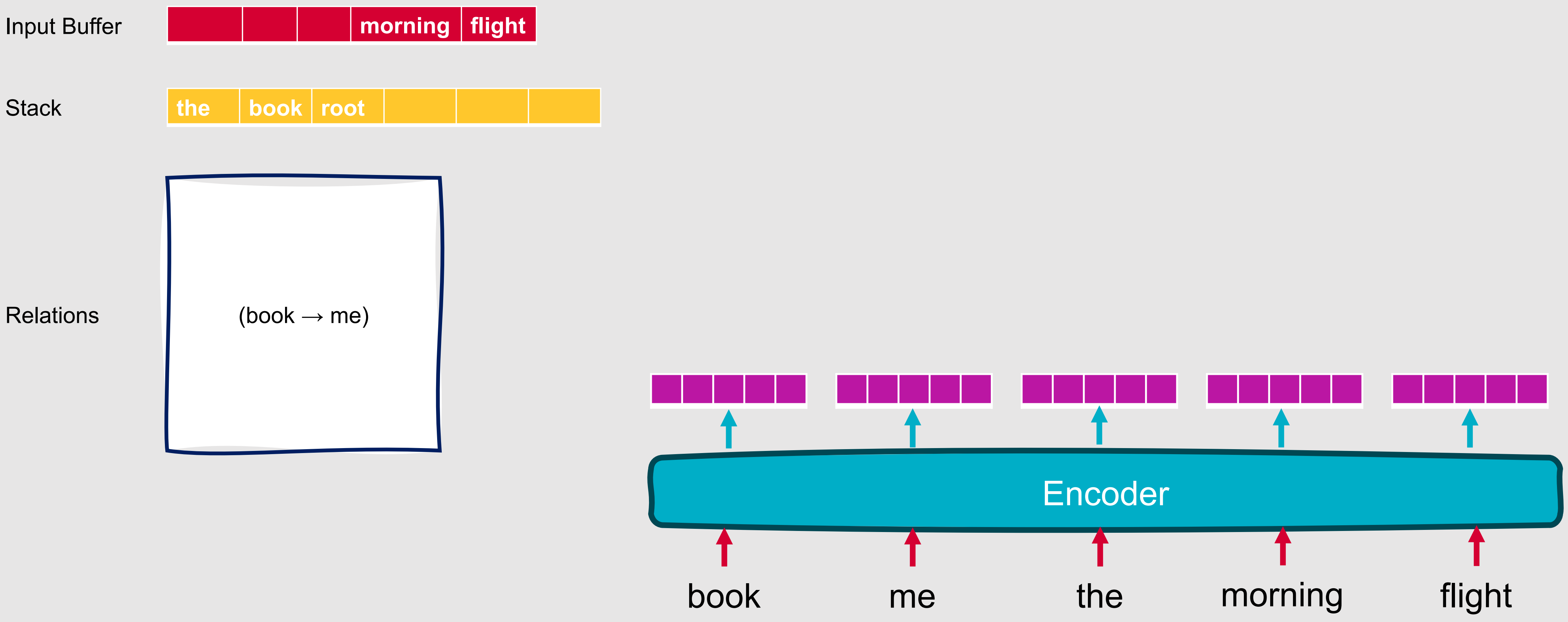
- Commonly:
  - Logistic regression
  - Support vector machines
- Recently:
  - Neural networks

Neural Network-based Oracle



Input Buffer				morning	flight	
Stack	the	book	root			
Relations	(book → me)					

Neural Network-based Oracle



Input Buffer

morning flight

Stack

the book root

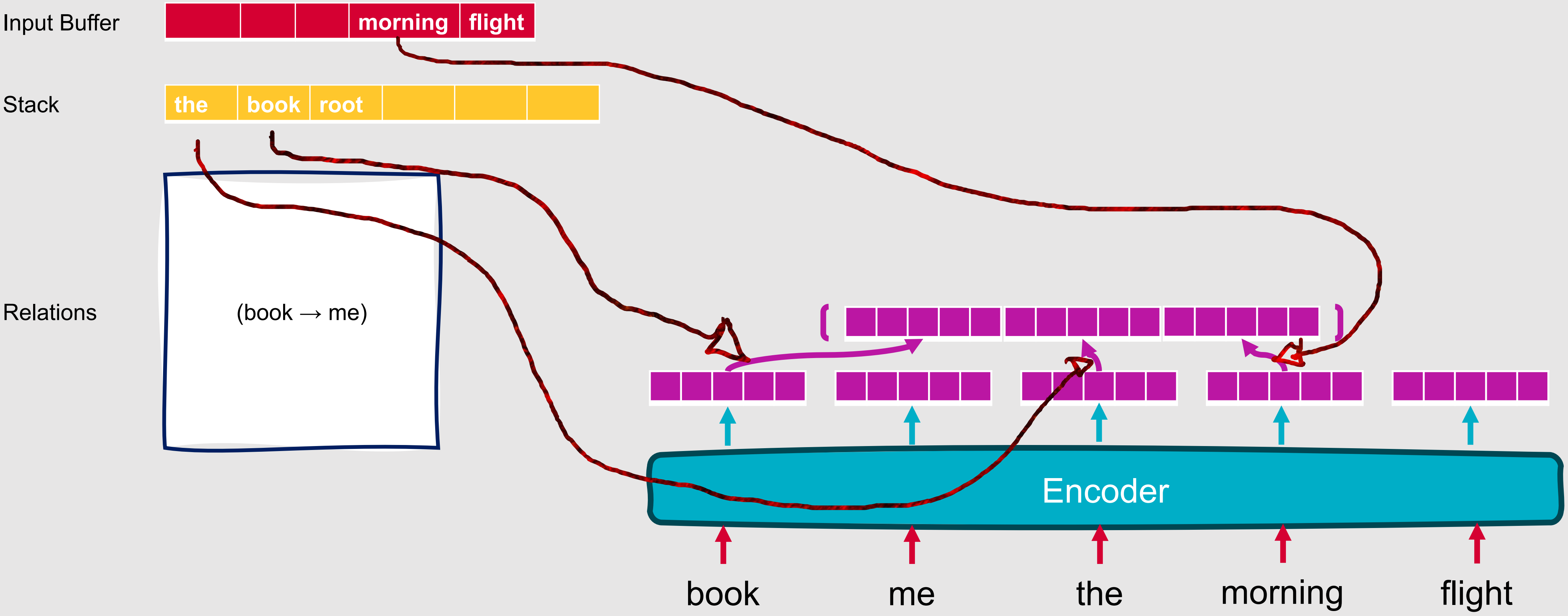
Relations

(book → me)

Encoder

book me the morning flight

Neural Network-based Oracle

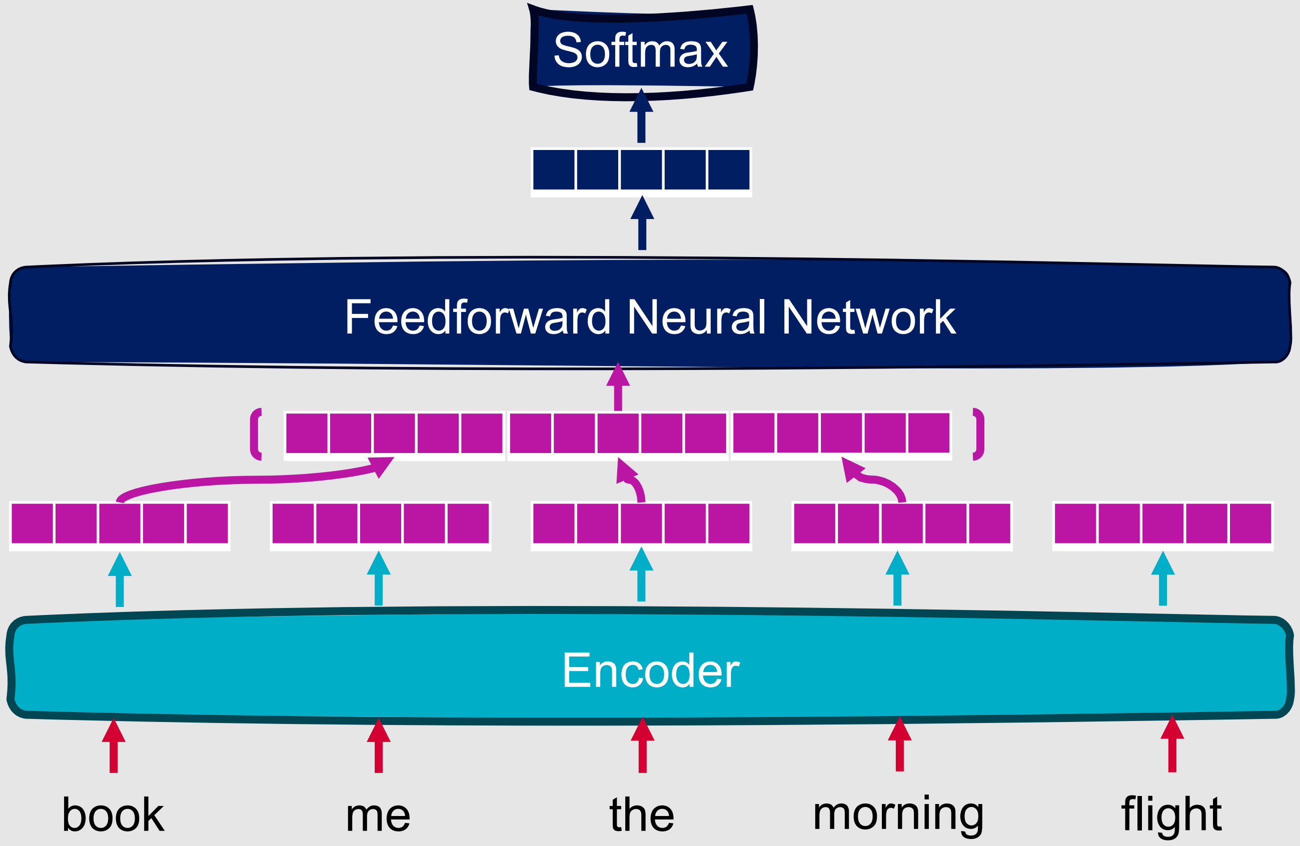


Neural Network-based Oracle

Input Buffer morning flight

Stack the book root

Relations (book → me)



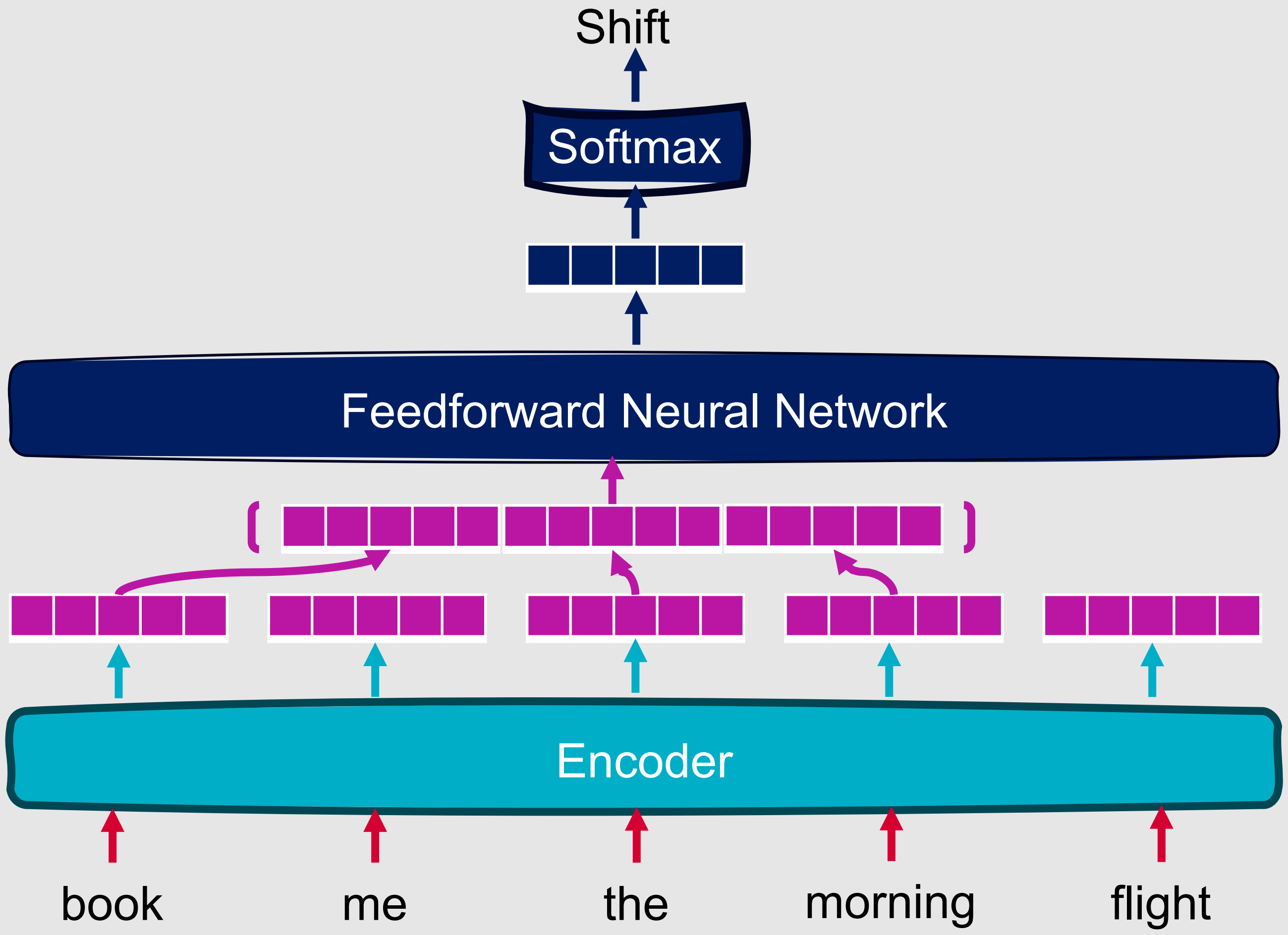
Neural Network-based Oracle

Input Buffer | | | | morning | flight

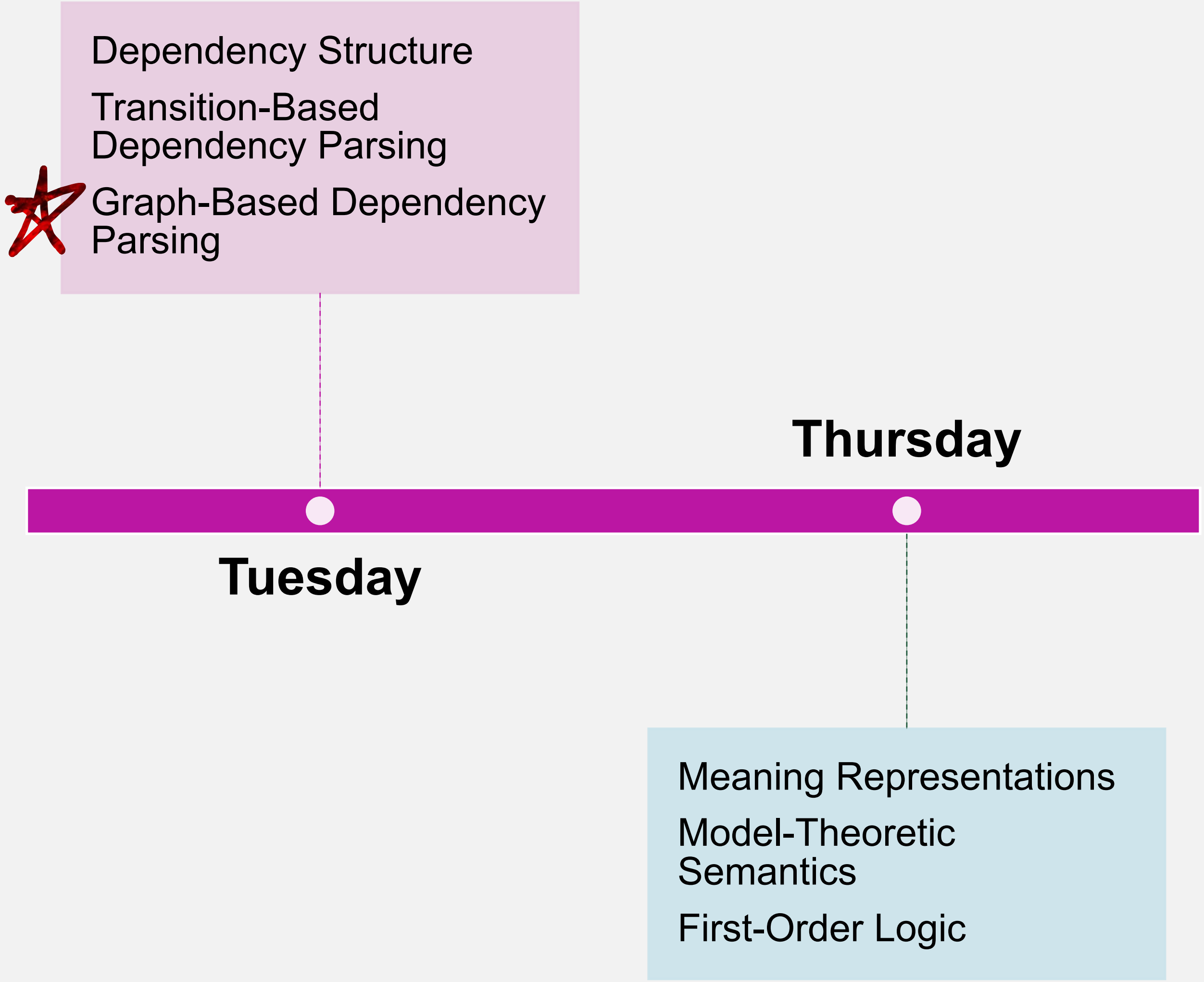
Stack | the | book | root | | |

Relations

(book → me)



This Week's Topics



Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Tuesday

Thursday

Meaning Representations

Model-Theoretic Semantics

First-Order Logic

Graph-based Dependency Parsing

- Search through the space of possible dependency trees for a given sentence, attempting to maximize some score
- This score is generally a function of the scores of individual subtrees within the overall tree
- **Edge-factored approaches** determine scores based on the scores of the edges that comprise the tree
  - overall_score(*t*) = $\sum_{e \in t} score(e)$
  - Letting *t* be a tree for a given sentence, and *e* be its edges

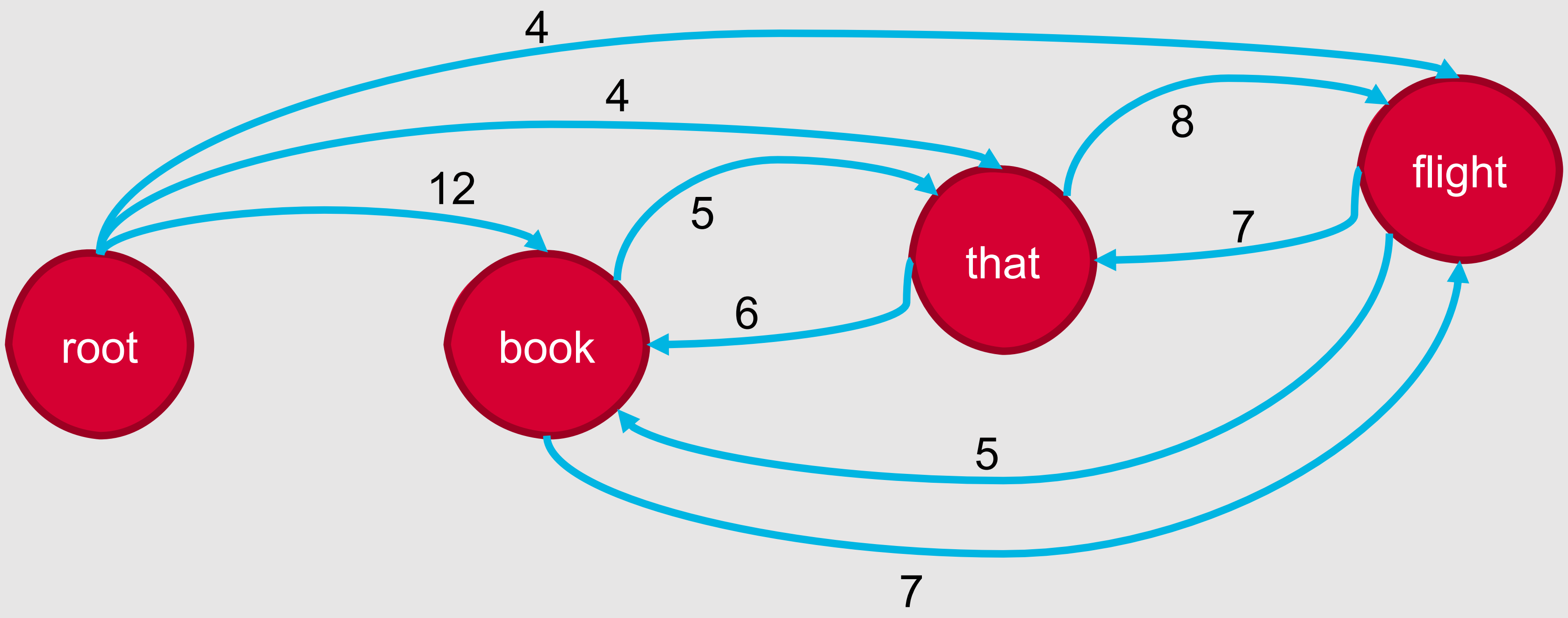
Why use graph-based methods for dependency parsing?

- Transition-based methods tend to have high accuracy for shorter dependency relations, but lower accuracy as the distance between words increases
- This is largely because transition-based methods are greedy (they can be fooled by seemingly-optimal local solutions)
- Graph-based methods score entire trees, thereby avoiding that issue

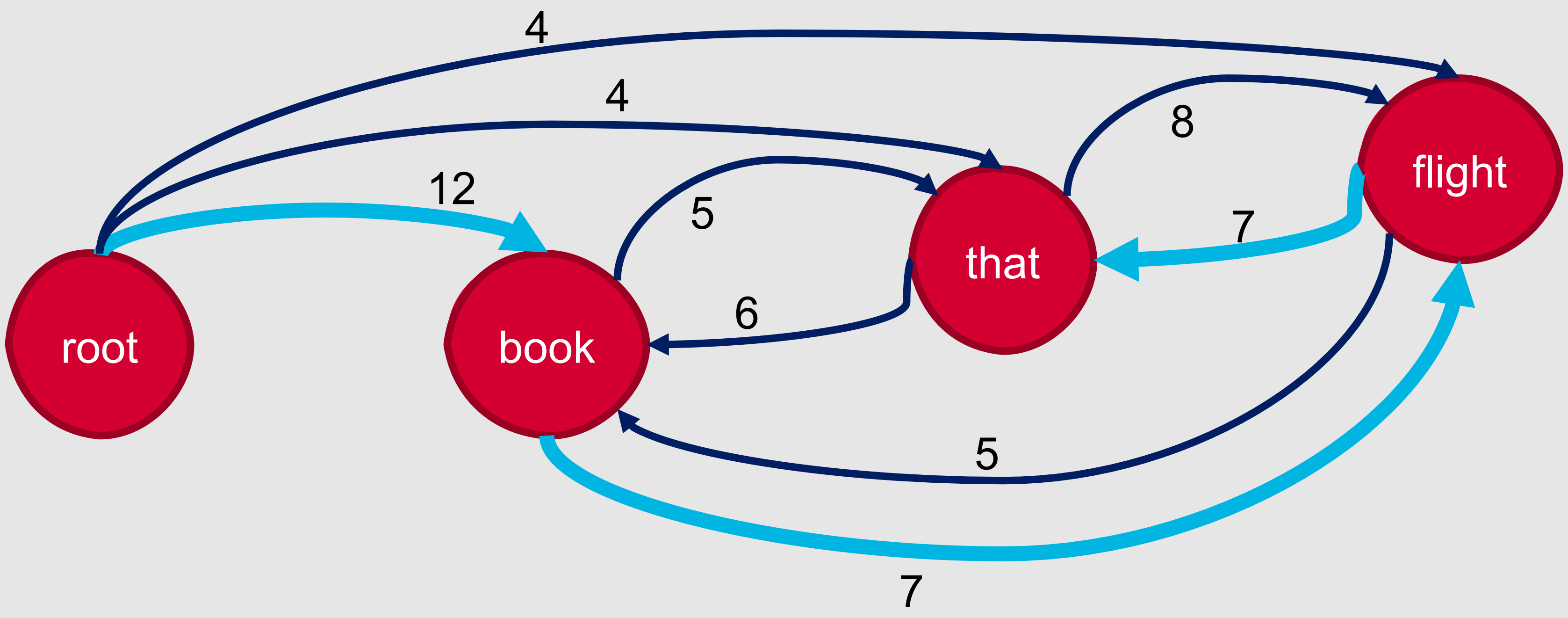
Maximum Spanning Tree

- Given an input sentence, construct a fully-connected, weighted, directed graph
  - Vertices are input words
  - Directed edges represent all possible head-dependent assignments
  - Weights reflect the scores for each possible head-dependent assignment, predicted by a supervised machine learning model
- A maximum spanning tree represents the preferred dependency parse for the sentence, as determined by the weights

Maximum Spanning Tree: Example

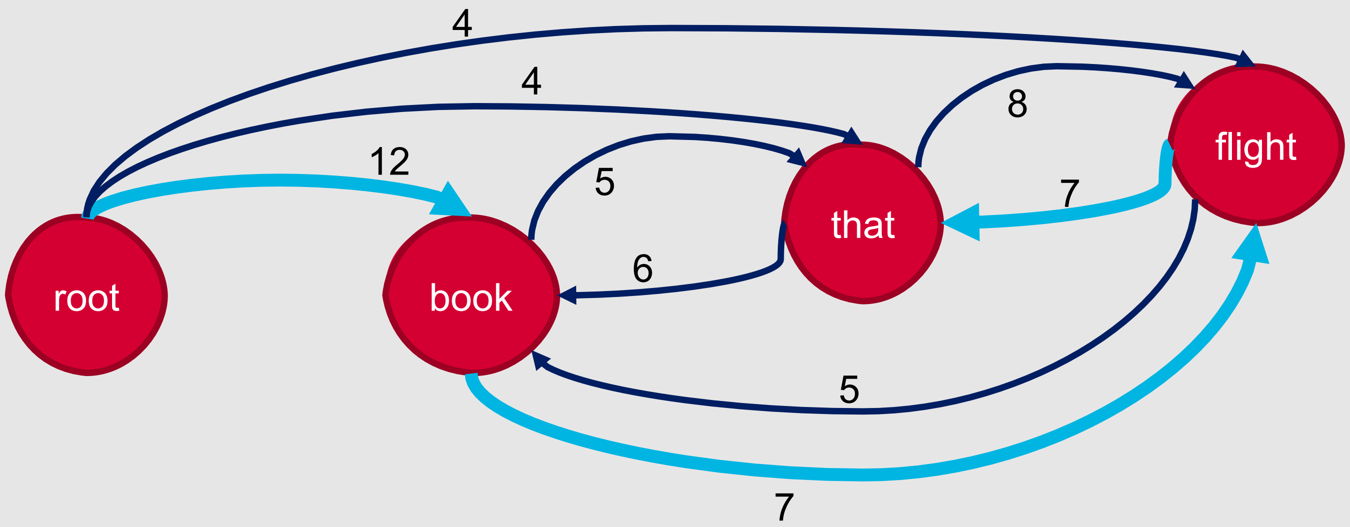


Maximum Spanning Tree: Example



Two things to keep in mind….

- Every vertex in a spanning tree has exactly one incoming edge
- Absolute values of the edge scores are not critical
  - Relative weights of the edges *entering* a vertex are what matter!



How do we know that we have a spanning tree?

- Given a fully-connected graph G = (V, E), a subgraph T = (V, F) is a spanning tree if:
  - It has no cycles
  - Each vertex (except the root) has exactly one edge entering it

How do we know that we have a *maximum* spanning tree?

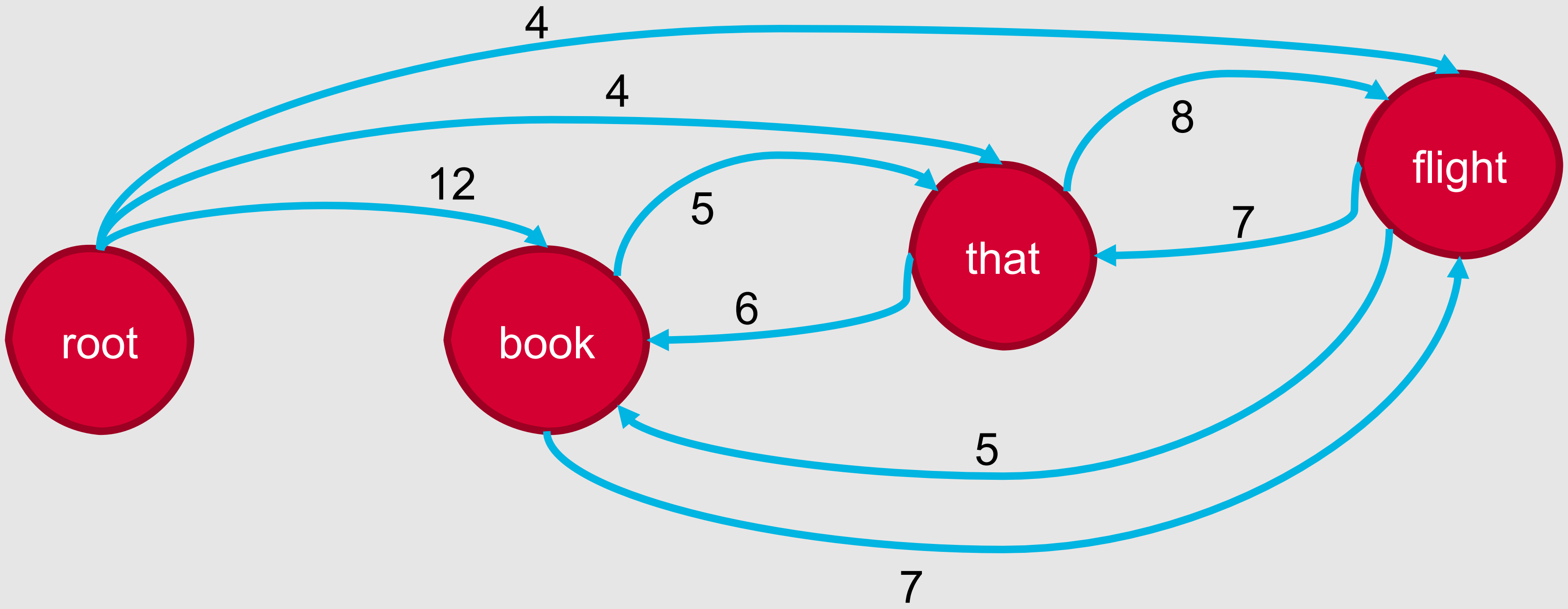
- **If the greedy selection process produces a spanning tree, then that tree is the maximum spanning tree**
- However, the greedy selection process may select edges that result in cycles
- If this happens, we can:
    - Collapse cycles into new nodes, with edges that previously entered or exited the cycle now entering or exiting the new node
    - Recursively apply the greedy selection process to the updated graph until a (maximum) spanning tree is found

Formal Algorithm

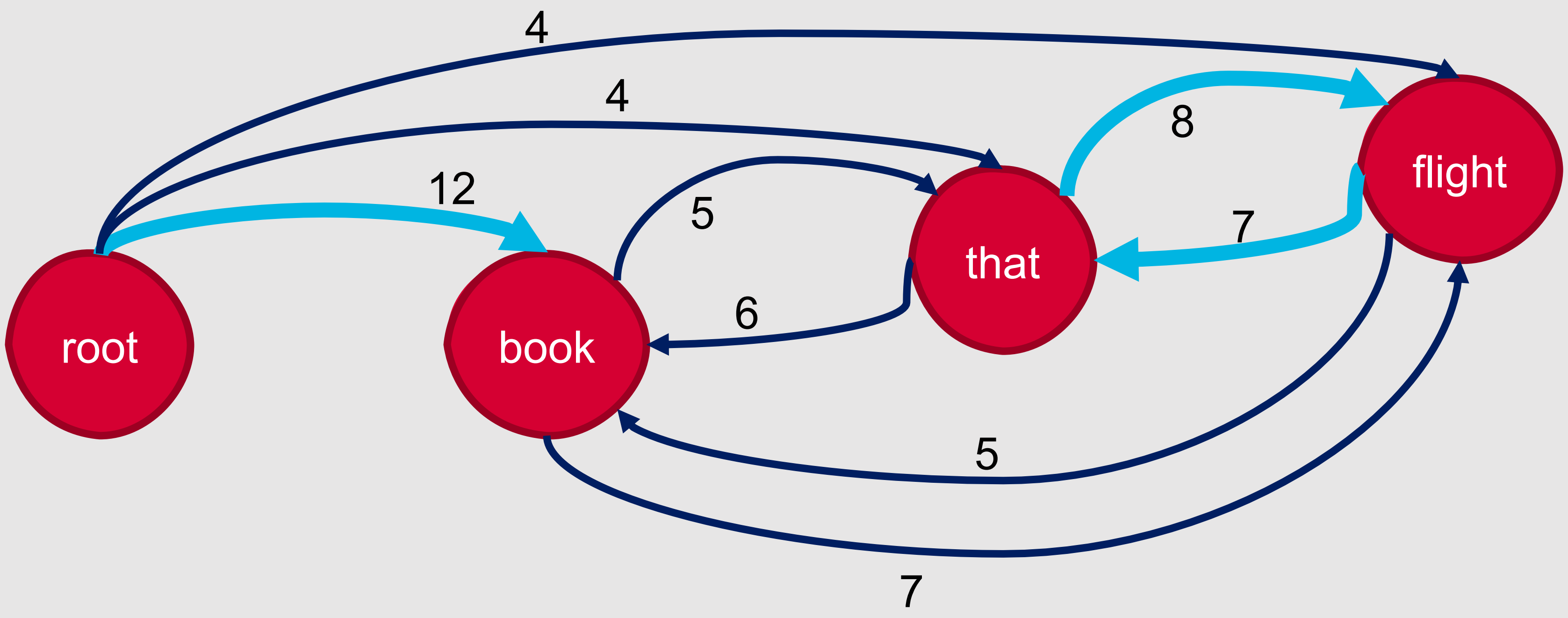
```
F ← []
T ← []
score’ ← []
for each v in V do:
    bestInEdge ← $\underset{e=(u,v)\in E}{\operatorname{argmax}} score[e]$
    F ← F ∪ bestInEdge
    for each $e = (u, v) \in E$ do:
        score’[e] ← score[e] - score[bestInEdge]

    if T=(V,F) is a spanning tree:
        return T
    else:
        C ← a cycle in F
        G’ ← collapse(G, C)
        T’ ← maxspanningtree(G’, root, score’) # Recursively call the current function
        T ← expand(T’, C)
        return T
```

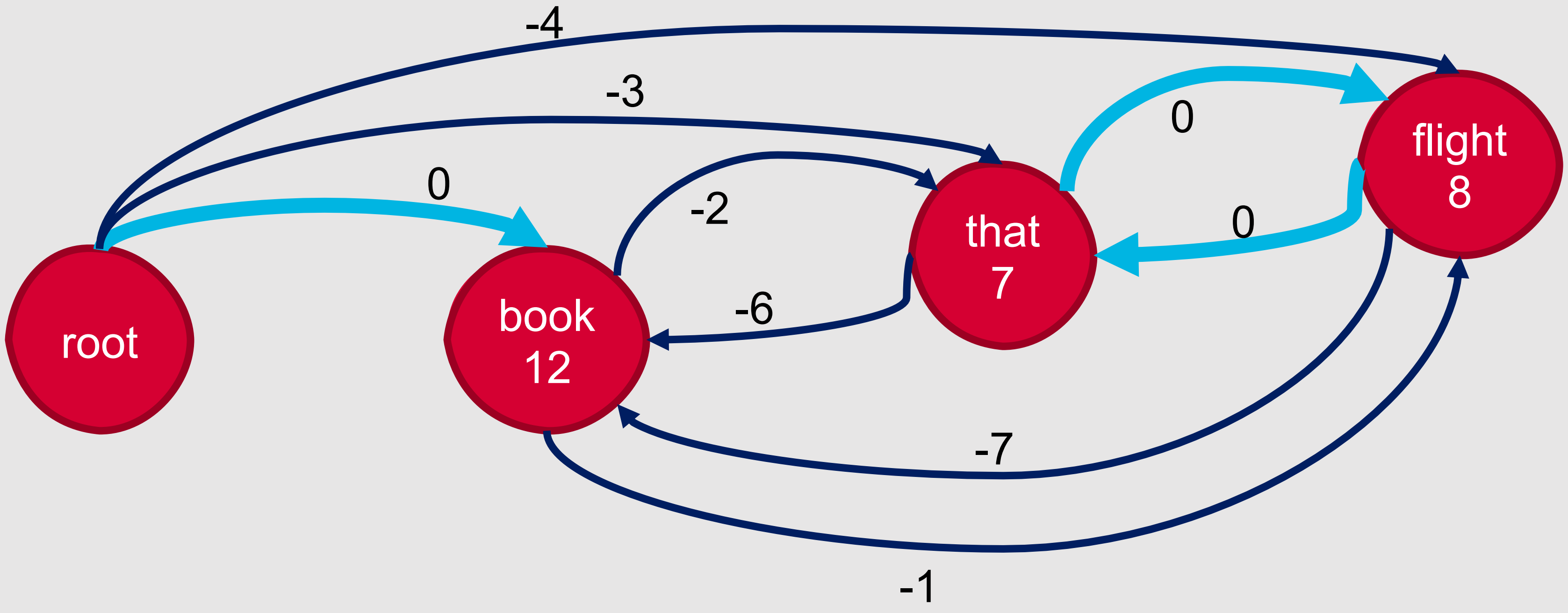
Maximum Spanning Tree: Updated Example



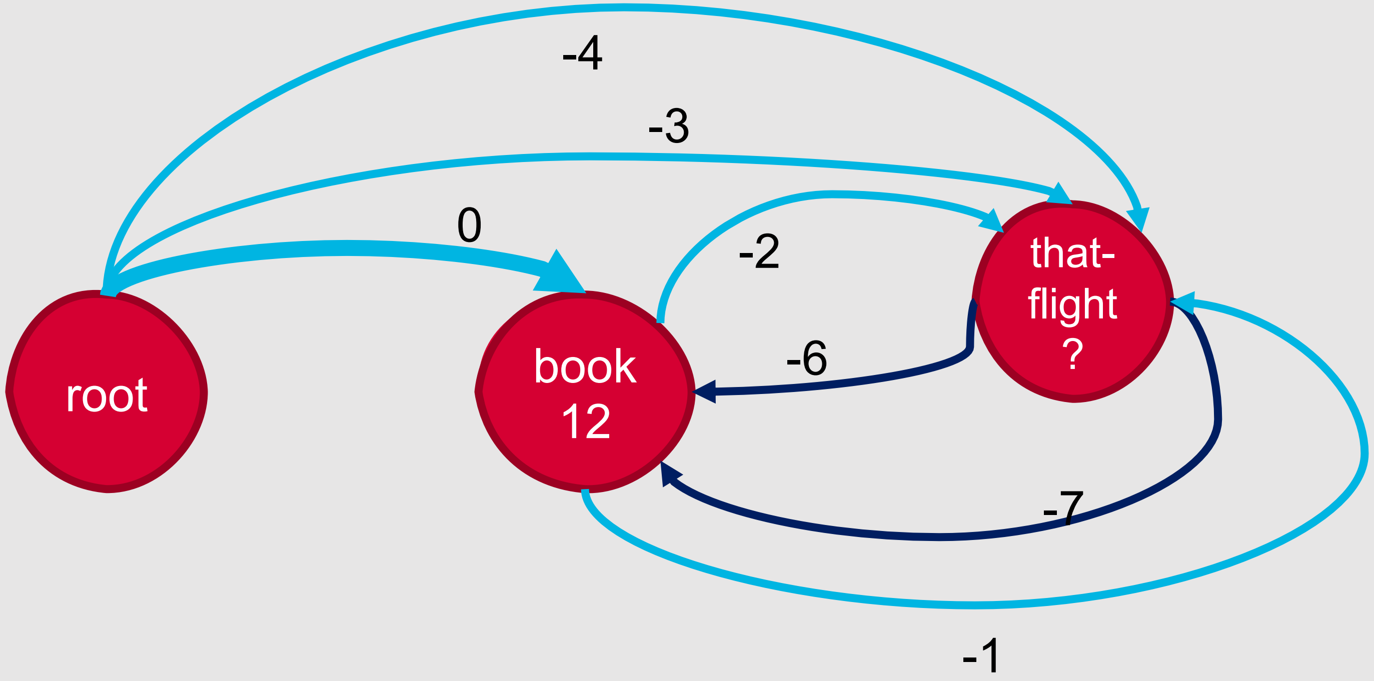
Maximum Spanning Tree: Updated Example



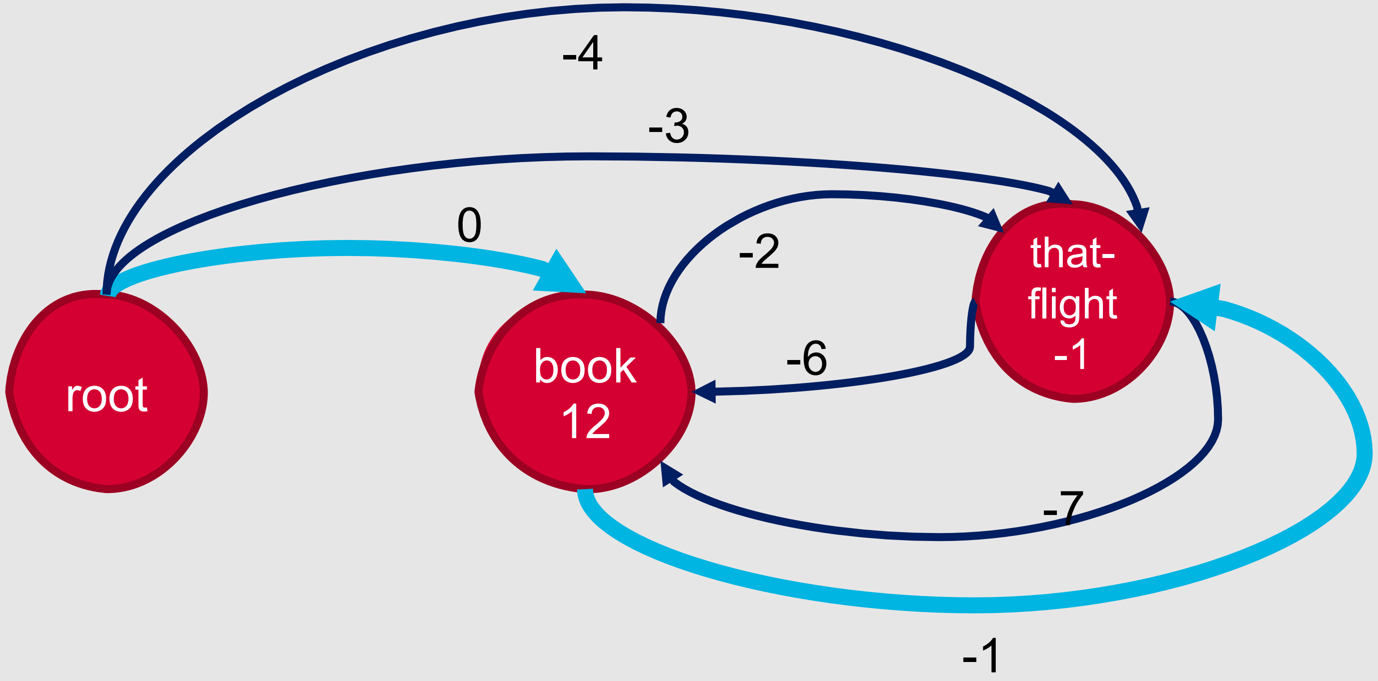
Maximum Spanning Tree: Updated Example



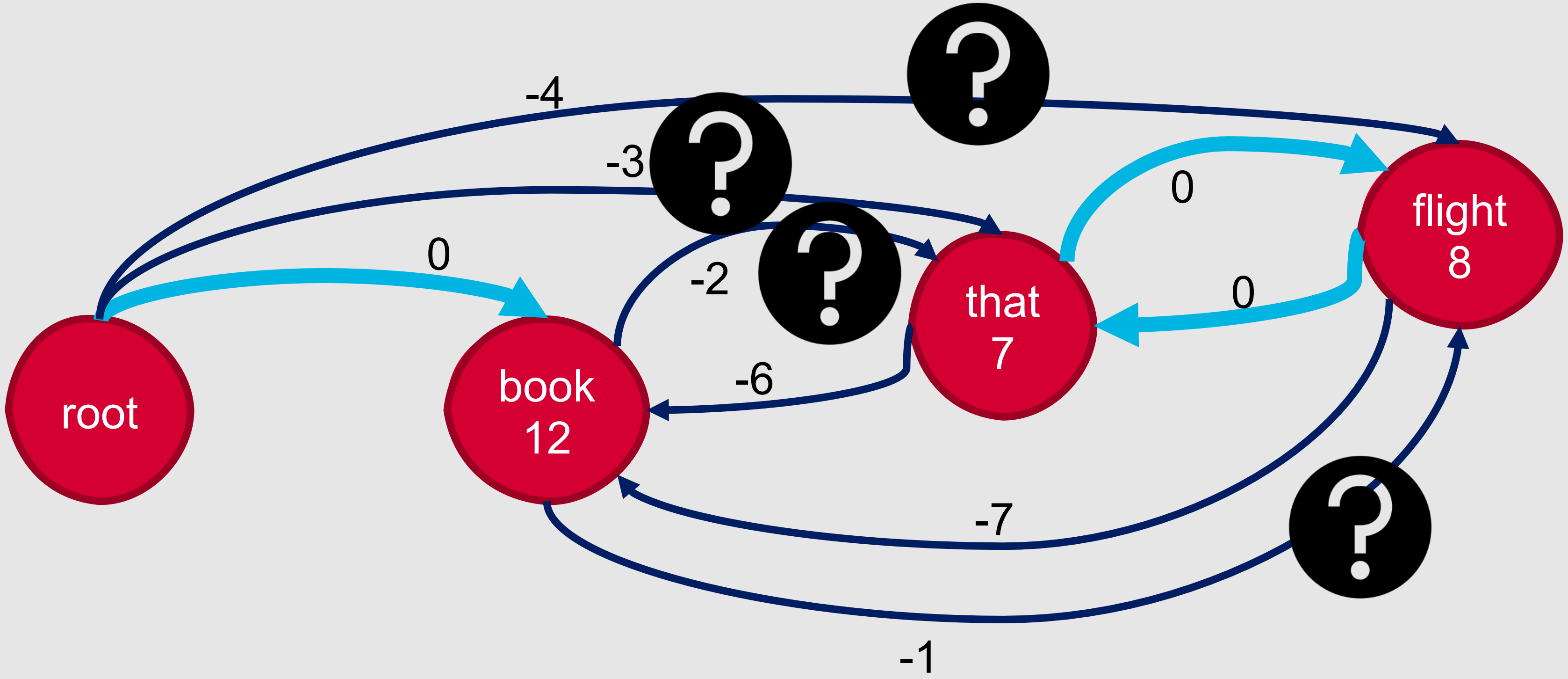
Maximum Spanning Tree: Updated Example



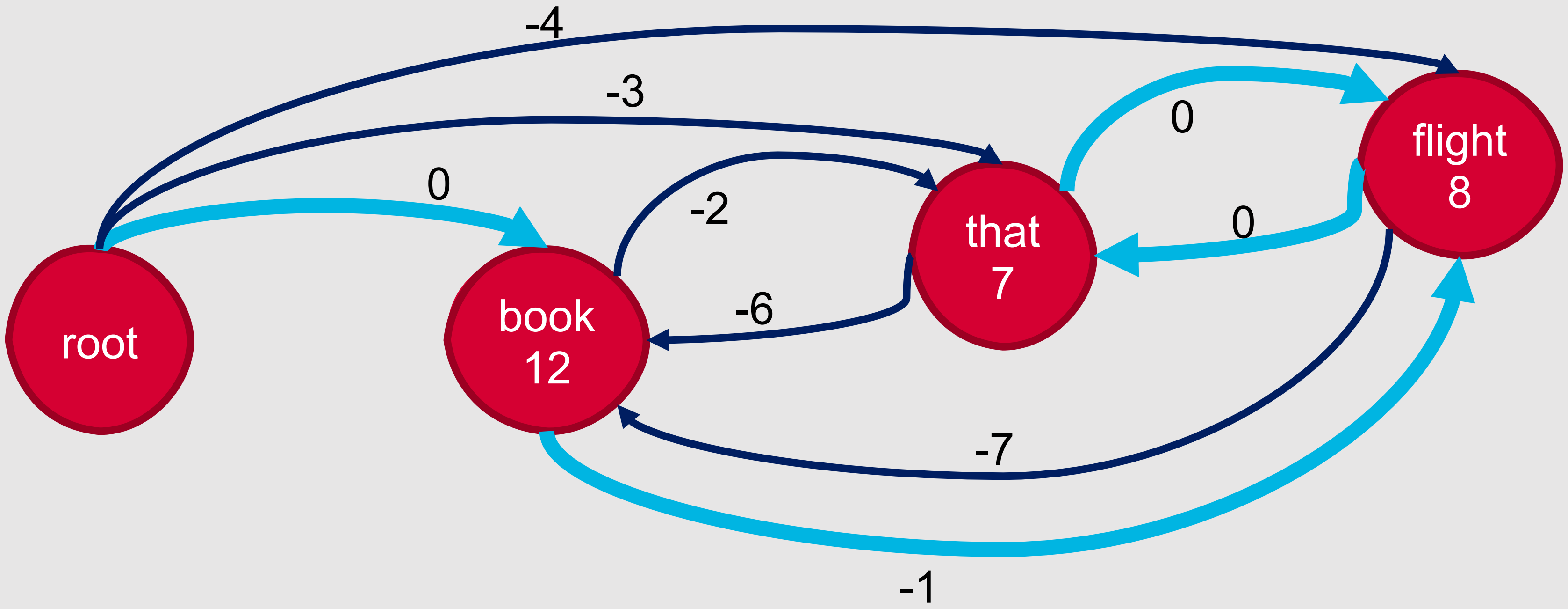
Maximum Spanning Tree: Updated Example



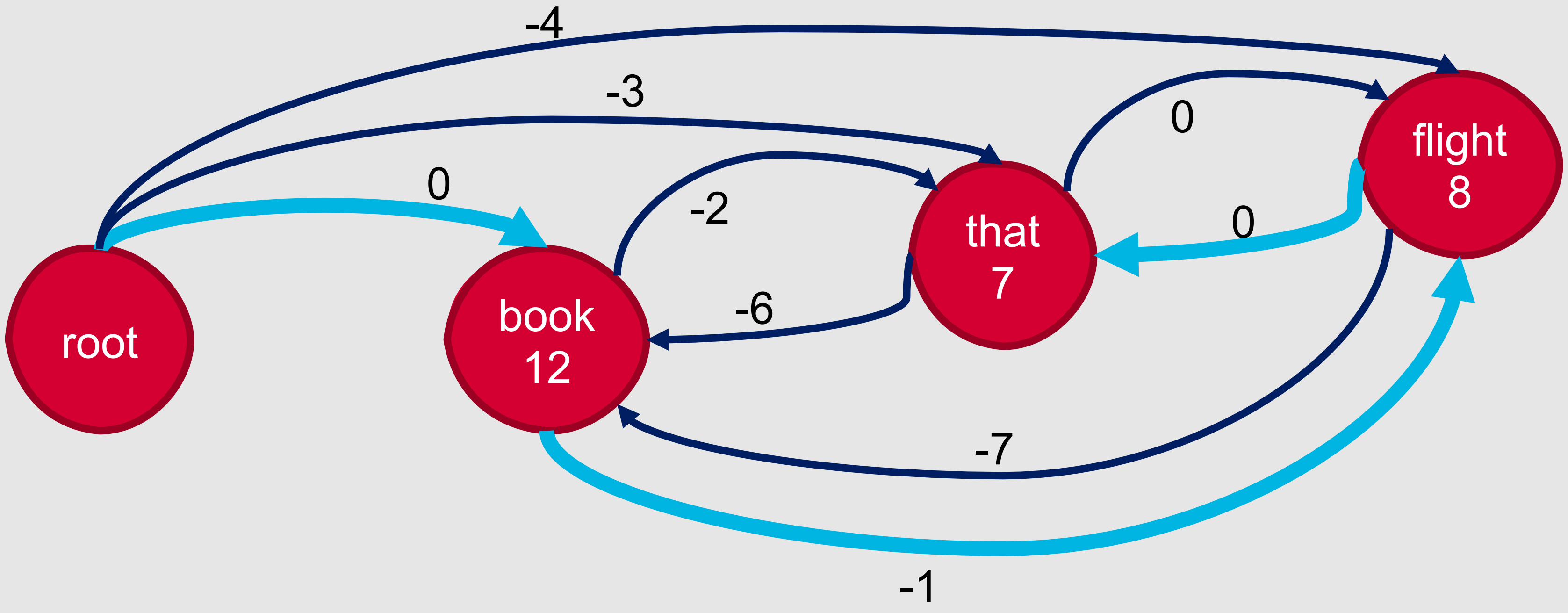
Maximum Spanning Tree: Updated Example



Maximum Spanning Tree: Updated Example



Maximum Spanning Tree: Updated Example



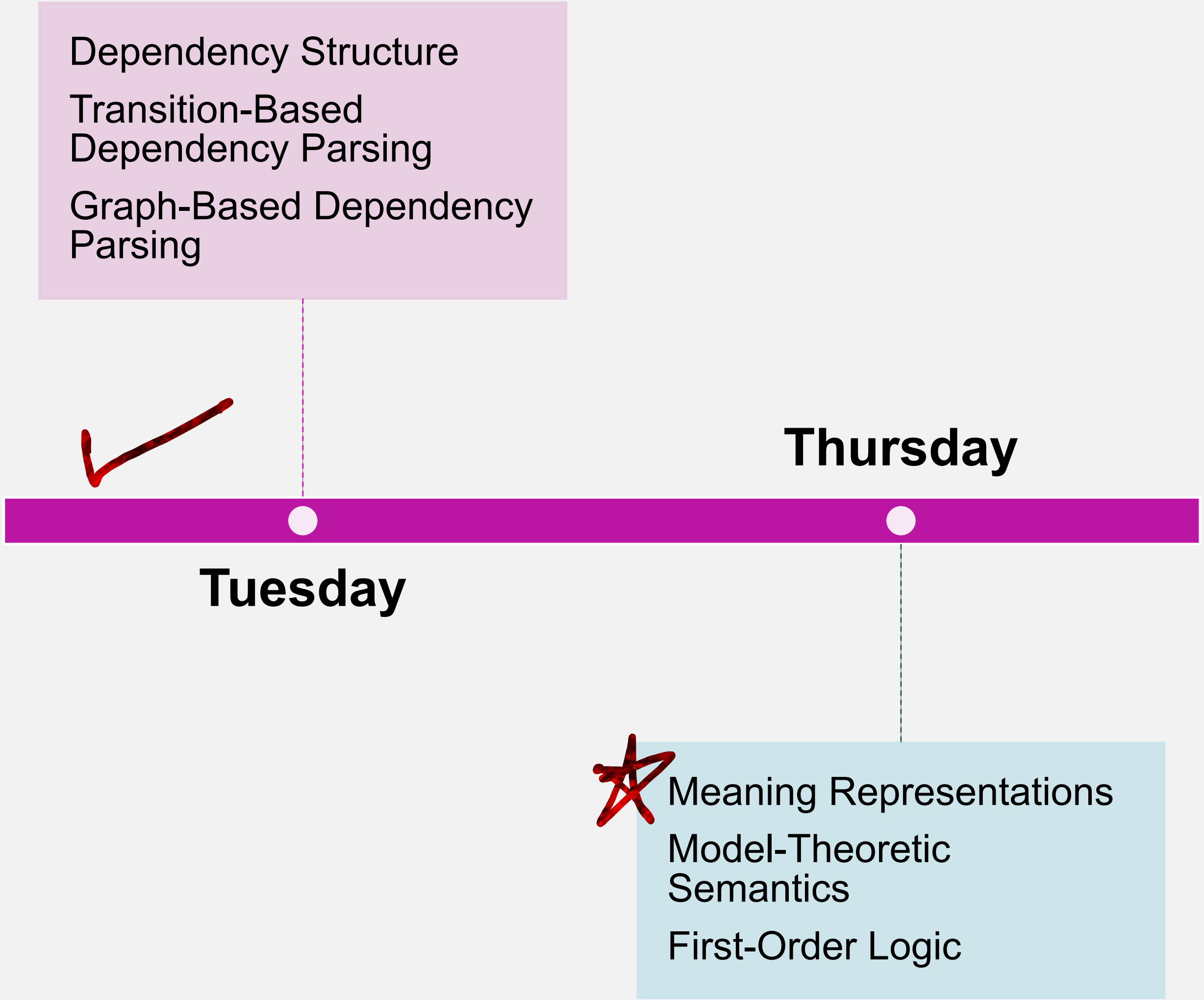
How do we train our model to predict edge weights?

- Similar approach to training the oracle in a transition-based parser
- Feature-based edge scoring models might predict weights based on:
  - Words, lemmas, parts of speech
  - Corresponding features from contexts before and after words
  - Word embeddings
  - Dependency relation type
  - Dependency relation direction
  - Distance from head to dependent
- We can also use neural networks for this process

Summary: Dependency Parsing

- **Dependency parsing** is the process of automatically determining **directed relationships between words** in a source sentence
- Many dependency tagsets exist, but currently the most common tagset is the set of **universal dependencies**
- Dependency parsers can be **transition-based** or **graph-based**
- A popular transition-based method is the **arc standard** approach
- A popular graph-based method is the **maximum spanning tree** approach
- Both make use of **supervised machine learning** to aid the decision-making process

This Week's Topics



Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Tuesday

Thursday

Meaning Representations

Model-Theoretic Semantics

First-Order Logic

Why do we need meaning representations?

- Somehow, we need to bridge the gap between **linguistic input** and **world knowledge** to perform semantic processing tasks such as:
  - Answering essay questions on exams
  - Deciding what to order at a restaurant
  - Detecting sarcasm
  - Following recipes

Logical Representations of Meaning

- **Goal: Represent commonsense world knowledge in logical form**
- There are many ways to represent meaning:
  - First-Order Logic
  - Semantic networks
  - Conceptual dependencies
  - Frame-based representations
  - All of these approaches assume that meaning representations consist of **structures** composed of **symbols**
    - **Symbols:** Representational vocabulary

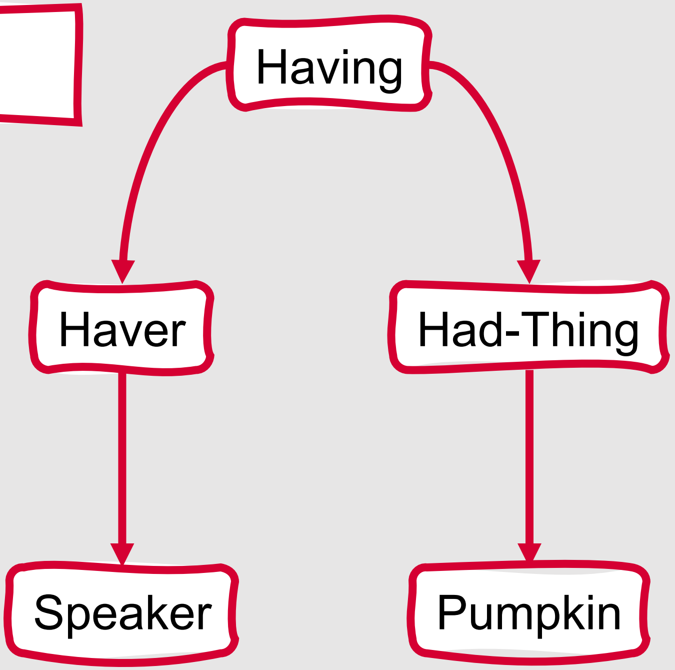
Sample Meaning Representations

I have a pumpkin.

$$\exists x, y\ \mathrm{Having}(x) \land\ \mathrm{Haver}(x, Speaker) \land \mathrm{HadThing}(x, y) \land \mathrm{Pumpkin}(y)$$

Having

Haver:	Speaker
HadThing:	Pumpkin



Symbols

- Correspond to **objects**, **properties** of objects, and **relations** among objects
- Symbols link linguistic input (words) to meaning (world knowledge)

Having

Haver:	Speaker
HadThing:	Pumpkin

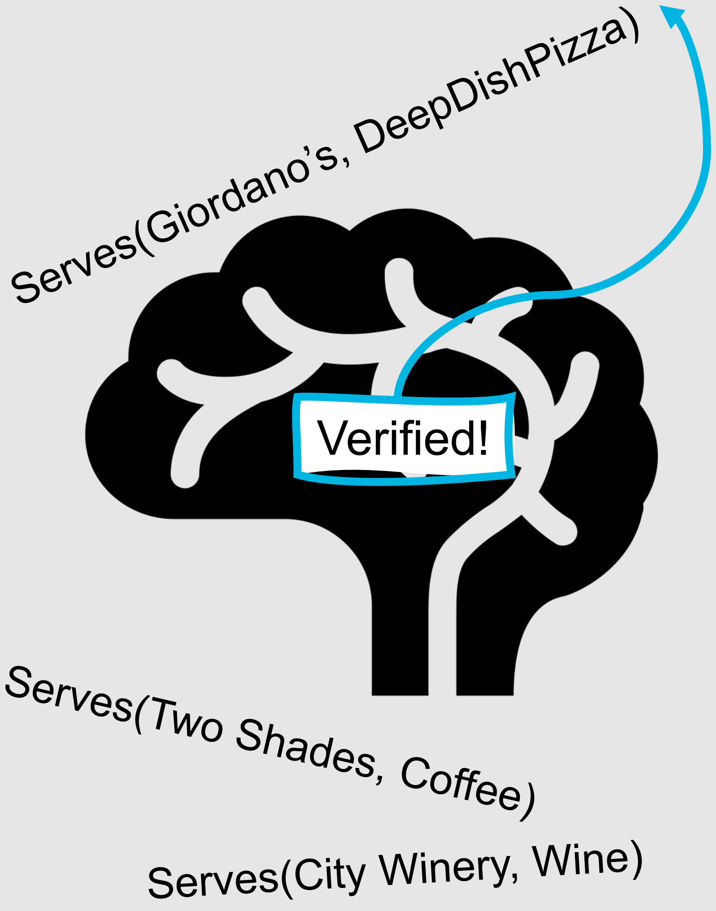
Meaning representations should be….

- Verifiable
- Unambiguous
- Able to map to a canonical form
- Supportive of inference and variables
- Expressive

Verifiability

- Meaning representations determine the relationship between (a) **the meaning of a sentence** and (b) **the world as we know it**
- Computational systems can verify the truth of a meaning representation for a sentence by matching it with **knowledge base** representations
  - **Knowledge Base:** A source of information about the world

Verifiability


Serves(Giordano's, DeepDishPizza)
Verified!
Serves(Two Shades, Coffee)
Serves(City Winery, Wine)

- Example proposition: **Giordano's serves deep dish pizza**.
- We can represent this as: **Serves(Giordano's, DeepDishPizza)**
- To verify the truth of this proposition, we would search a knowledge base containing facts about restaurants
- If we found a fact matching this, we have verified the proposition
- If not, we must assume that the fact is incorrect or, at best, our knowledge base is incomplete

Unambiguous Representations

- Ambiguity does not stop at syntax!
- Semantic ambiguities are everywhere:
  - Sarcasm
  - Idiom
  - Metaphor
  - Hyperbole
- Expressions may have different meaning representations depending on the circumstances in which they occur

Unambiguous Representations

Let’s eat somewhere near SEO.

Let’s eat somewhere near SEO.

- Ambiguities arising from figurative language require advanced solutions, but many semantic ambiguities can also arise from literal expressions
- To resolve semantic ambiguities, computational methods must select which from a set of possible interpretations is most correct, given the circumstances surrounding the linguistic input

Let’s devour some building near SEO!

Let’s eat at a restaurant near SEO!

Vagueness

- Closely related to ambiguity
- However, vagueness does not give rise to multiple representations
- In fact, it is advantageous for meaning representations to maintain a certain level of vagueness
    - Otherwise, you may be "overfitting" to your set of example sentences

Cookies?
Cake?
Ice cream?
Pie?
I want to eat dessert.

Canonical Form

- Sentences are ambiguous when they could reasonably be assigned multiple meaning representations
- However, **multiple sentences could also be assigned the same meaning representation**
  - Giordano’s serves deep dish pizza.
  - They have deep dish pizza at Giordano’s.
  - Deep dish pizza is served at Giordano’s.
  - You can eat deep dish pizza at Giordano’s.

Inference and Variables

- It's impossible for a knowledge base to comprehensively cover all facts about the world, so computational systems also need to be able to draw commonsense inferences based on meaning representations
  - **Will people who like deep dish pizza want to eat at Giordano's?**
    - We don't have a fact explicitly specifying that they do, but we can infer that if they like deep dish pizza, they will probably like a restaurant that serves it

Inference

- **Inference:** A system’s ability to draw valid conclusions based on the meaning representations of inputs and its store of background knowledge
- Systems must be able to draw conclusions about the truth of propositions that are not explicitly represented in the knowledge base but that are logically derivable from the propositions that are present

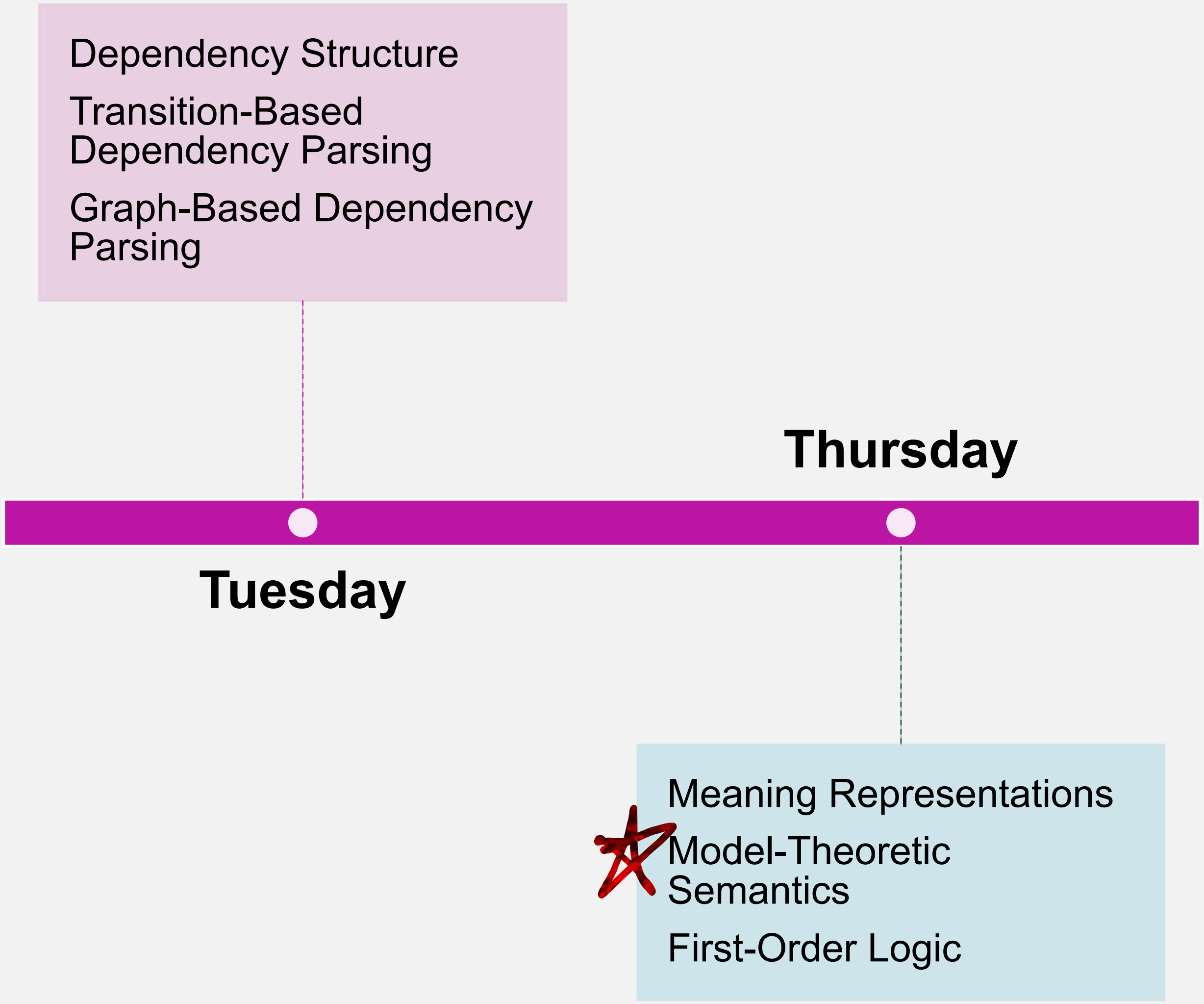
Variables

- Variables allow you to build propositions without requiring a specific instance of something
  - Serves(x, DeepDishPizza)
- These propositions can only be successfully matched by known instances from a knowledge base that would resolve in a truthful entire proposition
  - Serves(x, DeepDishPizza)
    - Serves(Giordano’s, DeepDishPizza) 🙂
    - Serves(Two Shades, DeepDishPizza) 🤨

Expressiveness

- **Expressive power:** The breadth of ideas that can be represented in a language
- Meaning representations must be **expressive** enough to handle a wide range of subject matter

This Week's Topics



Tuesday

Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Thursday

Meaning Representations

Model-Theoretic Semantics

First-Order Logic

Model-Theoretic Semantics

What do most meaning representation schemes share in common?

- An ability to represent objects, properties of objects, and relations among objects

A **model** is a formal construct that stands for a particular state of affairs in the world that we're trying to represent

Expressions (words or phrases) in the meaning representation language can be mapped to elements of the model

Relevant Terminology

- Vocabulary
  - **Non-Logical Vocabulary:** Open-ended sets of names for objects, properties, and relations in the world we're representing
  - **Logical Vocabulary:** Closed set of symbols, operators, quantifiers, and links that provide the formal means for composing expressions in the language
- **Domain:** The set of objects that are part of the state of affairs being represented in the model
- **Each object in the non-logical vocabulary corresponds to a unique element in the domain**; however, each element in the domain does not need to be mentioned in a meaning representation

Additional Terminology

- For a given domain, **objects** are elements
  - grapes, violets, plums, CS421, Usman, Eli
- **Properties** are sets of elements corresponding to a specific characteristic
  - purple = {grapes, violets, plums}
- **Relations** are sets of tuples, each of which contain domain elements that take part in a specific relation
  - TAFor = {(CS421, Usman), (CS421, Eli)}

Functions

- We create mappings from non-logical vocabulary to formal denotations using **functions** or interpretations
- Assume that we have:
  - A collection of restaurant patrons and restaurants
  - Various facts regarding the likes and dislikes of patrons
  - Various facts about the restaurants
- In our current state of affairs (our **model**) we're concerned with four patrons designated by the non-logical symbols (**elements**) *Natalie*, *Devika*, *Nikolaos*, and *Mina*
- We'll use the constants *a*, *b*, *c*, and *d* to refer to those respective elements

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

- We're also concerned with three restaurants designated by the non-logical symbols *Giordano's*, *IDOF*, and *Artopolis*
- We'll use the constants *e*, *f*, and *g* to refer to those respective elements

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

- Finally, we’ll assume that our model deals with three cuisines in general, designated by the non-logical symbols *Italian*, *Mediterranean*, and *Greek*
- We’ll use the constants *i*, *j*, and *k* to refer to those elements

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

- Now, let’s assume we need to represent a few properties of restaurants:
  - *Fast* denotes the subset of restaurants that are known to make food quickly
  - *TableService* denotes the subset of restaurants for which a waiter will come to your table to take your order
- We also need to represent a few relations:
  - *Like* denotes the tuples indicating which restaurants individual patrons like
  - *Serve* denotes the tuples indicating which restaurants serve specific cuisines

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

- This means that we have created the domain D = {a, b, c, d, e, f, g, i, j, k}
- We can evaluate representations like *Natalie likes IDOF* or *Giordano’s serves Greek* by mapping the objects in the meaning representations to their corresponding domain elements, and any links to the appropriate relations in the model
  - Natalie likes IDOF → a likes f → Like(a, f) 🙂
  - Giordano’s serves Greek → e serves k → Serve(e, k) 🤨

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano's, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

- Thus, we're just using sets and operations on sets to ground the expressions in our meaning representations
- What about more complex sentences?
  - Nikolaos likes Giordano's and Devika likes Artopolis.
  - Mina likes fast restaurants.
  - Not everybody likes IDOF.

Example Application

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

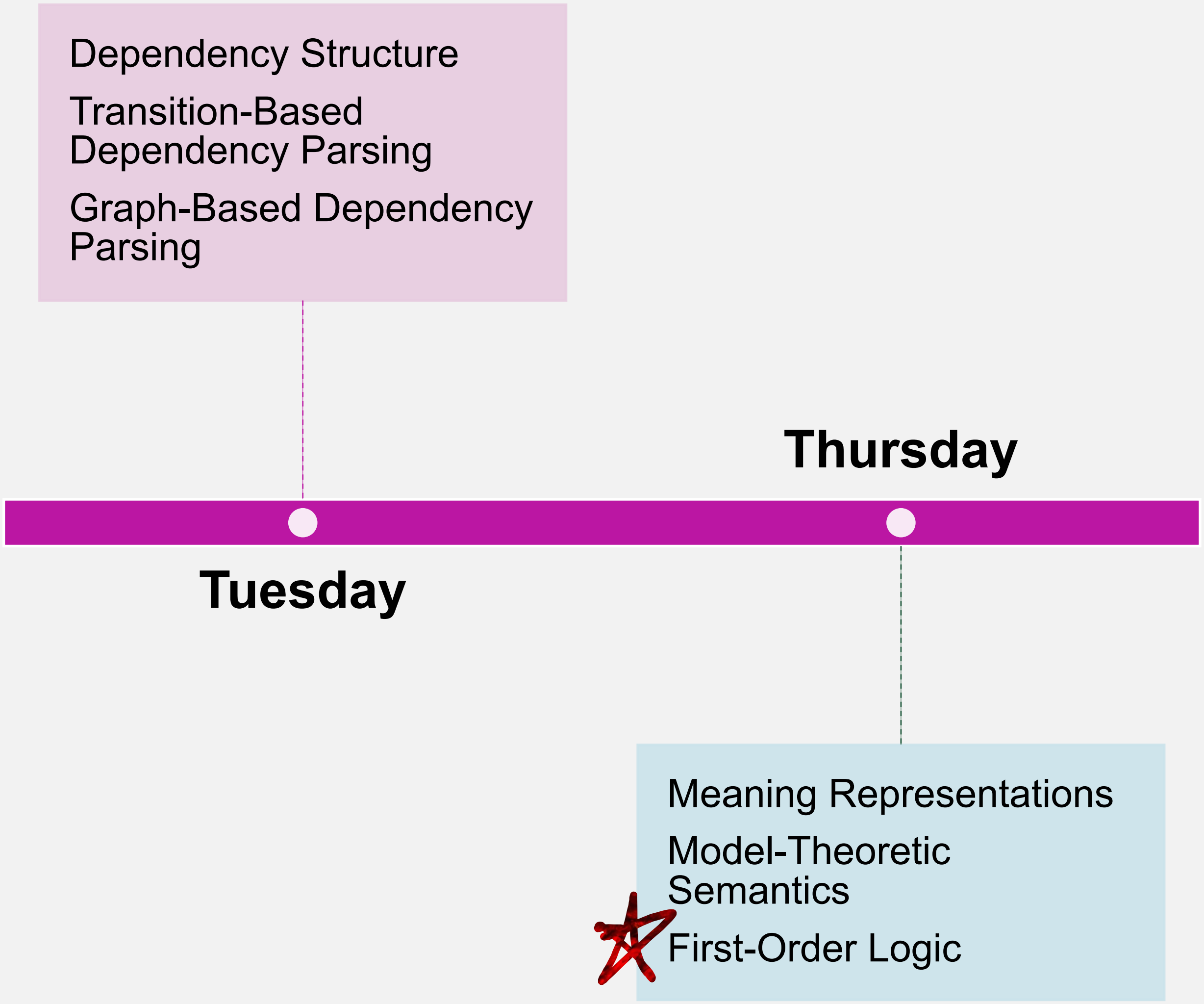
restaurants = {Giordano's, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

- Plausible meaning representations for the previous examples will not map directly to individual entities, properties, or relations!
- They involve:
  - Conjunctions
  - Equality
  - Variables
  - Negations
- What we need are **truth-conditional semantics**
- This is where **first-order logic** is useful

This Week's Topics



Dependency Structure

Transition-Based Dependency Parsing

Graph-Based Dependency Parsing

Tuesday

Thursday

Meaning Representations

Model-Theoretic Semantics

★ First-Order Logic

What is first-order logic?

- A **meaning representation language** (a way to represent knowledge in a way that is computationally verifiable and supports semantic inference)

Elements of First-Order Logic

- **Term:** First-order logic device for representing objects
  - Constants
  - Functions
  - Variables
- Common across all types of terms:
  - Each one can be thought of as a way of pointing to a specific object

Elements of First-Order Logic

- **Constants:** Specific objects in the world being described
  - Conventionally depicted as single capitalized letters (A, B) or words (Natalie, Devika)
  - Refer to exactly one object, although objects can have more than one constant that refers to them
- **Functions:** Concepts that are syntactically equivalent to single-argument predicates
  - Can refer to specific objects without having to associate a named constant with them, e.g., LocationOf(Giordano’s)
- **Variables:** Provide the ability to make assertions and draw inferences without having to refer to a specific named object
  - Conventionally depicted as single lowercase letters

Basic Elements of First-Order Logic

- **Predicates:** Symbols that refer to the relations between a fixed number of objects in the domain
    - Can have one or more arguments
        - Serve(Giordano’s, Italian)
            - Relates two objects
        - Restaurant(Giordano’s)
            - Asserts a property of a single object
- Predicates can be put together using **logical connectives**
    - and ∧
    - or ∨
    - implies →
- They can also be negated
    - not ¬

Variables and Quantifiers

- Two basic operators in first-order logic are:
  - ∃: The existential quantifier
    - Pronounced “there exists”
  - ∀: The universal quantifier
    - Pronounced “for all”
- These two operators make it possible to represent many more sentences!
  - a restaurant → ∃x Restaurant(x)
  - all restaurants → ∀x Restaurant(x)

We can combine these operators with other basic elements of first-order logic to build logical representations of complex sentences.

- Nikolaos likes Giordano’s and Devika likes Artopolis.
  - Like(Nikolaos, Giordano’s) $\wedge$ Like(Devika, Artopolis)
- Mina likes fast restaurants.
  - $\forall x$ Fast(x) $\rightarrow$ Like(Mina, x)
- Not everybody likes IDOF.
  - $\exists x$ Person(x) $\wedge$ $\neg$Like(x, IDOF)

Semantics of First-Order Logic

- Symbols for objects, properties, and relations acquire meaning based on their correspondences to “real” objects, properties, and relations in the external world
- We define meaning based on truth-conditional mappings between expressions in a meaning representation and the state of affairs being modeled

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$
False	False	True	False	False	True
False	True	True	False	True	True
True	False	False	False	True	False
True	True	False	True	True	True

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Likes(Natalie, Giordano’s) ∧ Likes(Devika, Giordano’s)

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Likes(Natalie, Giordano’s) ∧ Likes(Devika, Giordano’s)

Likes(a, e) ∧ Likes(b, e)

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Likes(Natalie, Giordano’s) ∧ Likes(Devika, Giordano’s)

Likes(a, e) ∧ Likes(b, e)

☺

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Likes(Natalie, Giordano’s) ∧ Likes(Devika, Giordano’s)

Likes(a, e) ∧ Likes(b, e)

☺ ☹

Example: Is the following sentence valid according to our model?

patron = {Natalie, Devika, Nikolaos, Mina} = {a, b, c, d}

restaurants = {Giordano’s, IDOF, Artopolis} = {e, f, g}

cuisines = {Italian, Mediterranean, Greek} = {i, j, k}

Fast = {f}
TableService = {e, g}
Likes = {(a, e), (a, f), (a, g), (b, g), (c, e), (d, f)}
Serve = {(e, i), (f, j), (g, k)}

Natalie likes Giordano’s and Devika likes Giordano’s.

Likes(Natalie, Giordano’s) ∧ Likes(Devika, Giordano’s)

Likes(a, e) ∧ Likes(b, e)

☺ ☹

False …not valid!

A few additional notes….

- Formulas involving ∃ are true if there is *any* substitution of terms for variables that results in a formula that is true according to the model
- Formulas involving ∀ are true only if *all* substitutions of terms for variables result in formulas that are true according to the model

How do we infer facts not explicitly included in the knowledge base?

- **Modus ponens:** If a conditional statement is accepted (if p then q), and the **antecedent** (p) holds, then the **consequent** (q) may be inferred
- More formally:

$$\frac{\begin{array}{l}\alpha \\ a \Rightarrow \beta\end{array}}{\beta}$$

Example: Inference

$$\begin{array}{c} \text{GreekRestaurant}(Artopolis) \\ \forall x\ \text{GreekRestaurant}(x) \Rightarrow \text{Serves}(x, GreekFood) \\ \hline \text{Serves}(Artopolis, GreekFood) \end{array}$$

conditional statement accepted ✓

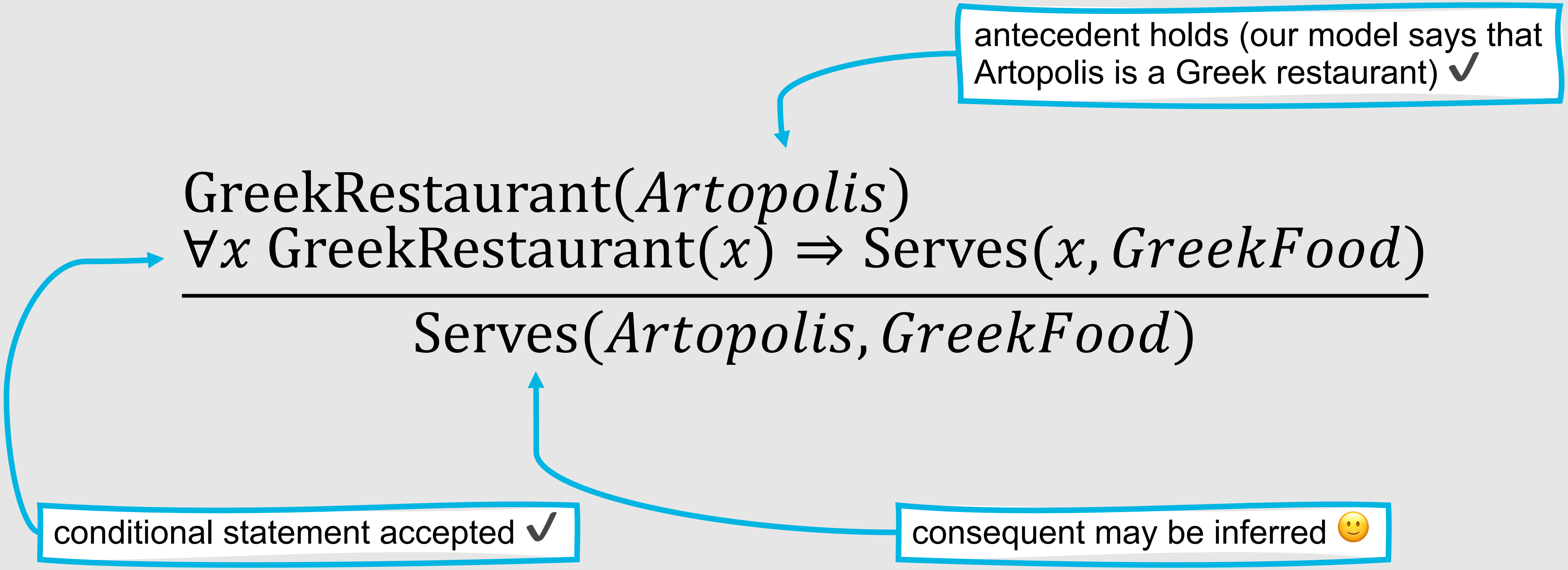
Example: Inference

antecedent holds (our model says that Artopolis is a Greek restaurant) ✓

$$\begin{array}{c}
\text{GreekRestaurant}(Artopolis) \\
\forall x\ \text{GreekRestaurant}(x) \Rightarrow \text{Serves}(x, GreekFood) \\
\hline
\text{Serves}(Artopolis, GreekFood)
\end{array}$$

conditional statement accepted ✓

Example: Inference



antecedent holds (our model says that Artopolis is a Greek restaurant) ✓

$$\frac{\begin{array}{l}\text{GreekRestaurant}(Artopolis)\\ \forall x\ \text{GreekRestaurant}(x) \Rightarrow \text{Serves}(x, GreekFood)\end{array}}{\text{Serves}(Artopolis, GreekFood)}$$

conditional statement accepted ✓

consequent may be inferred 🙂

Representing States and Events

States: Conditions or properties that remain unchanged over some period of time

Events: Indicate changes in some state of affairs

Events can be particularly challenging to represent in formal logic!

- You may need to:
  - Determine the correct number of roles for the event
  - Represent facts about different roles associated with the event
  - Ensure that all correct inferences can be derived directly from the event representation
  - Ensure that no incorrect inferences can be derived from the event representation
- Some events may theoretically take a variable number of arguments
  - Natalie drinks.
  - Natalie drinks tea.
- However, predicates in first-order logic have fixed **arity** (they accept a fixed number of arguments)

How do we deal with this?

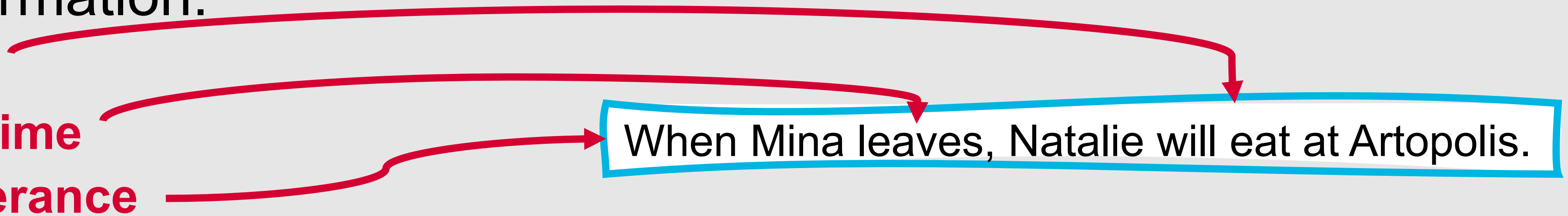
- Make as many different predicates as are needed to handle all of the different ways an event can behave
  - $Drink_1(Natalie)$
  - $Drink_2(Natalie, tea)$
  - Unfortunately, this can be costly (lots of different predicates would need to be stored for many words!)
- Another (also not-so-scalable) solution is to use **meaning postulates**
  - $\forall x,y\ Drink_2(x, y) \rightarrow Drink_1(x)$
- Finally, you can allow missing arguments
  - $\exists x\ Drink(Natalie, x)$
  - $Drink(Natalie, tea)$
  - Still not perfect …in the example case, it implies that one always has to be drinking a specific thing

Instead of regular variables, we can add event variables.

- **Event variable:** An argument to the event representation that allows for additional assertions to be included if needed
  - ∃e Drink(Natalie, e)
- If we determine that the actor must drink something specific: ∃e Drink(Natalie, e) ∧ Beverage(e, tea)
- More generally, we could define the representation:
  - ∃e Drink(e) ∧ Drinker(e, Natalie) ∧ Beverage(e, tea)
- With this change:
  - No need to specify a fixed number of arguments for a given surface predicate
  - Logical connections are satisfied without using meaning postulates

Ideally, meaning representations will also include information about time and aspect.

- Temporal information:
  - **Event time**
  - **Reference time**
  - **Time of utterance**



When Mina leaves, Natalie will eat at Artopolis.

- Aspectual information:
  - **Stative:** Event captures an aspect of the world at a single time point
    - Natalie knew what she wanted to eat.
  - **Activity:** Event occurs over some span of time
    - Natalie is eating.
  - **Accomplishment:** Event has a natural end point and results in a particular state
    - Natalie ate lunch at Artopolis.
  - **Achievement:** Event happens in an instant, but still results in a particular state
    - Natalie finished her meal.

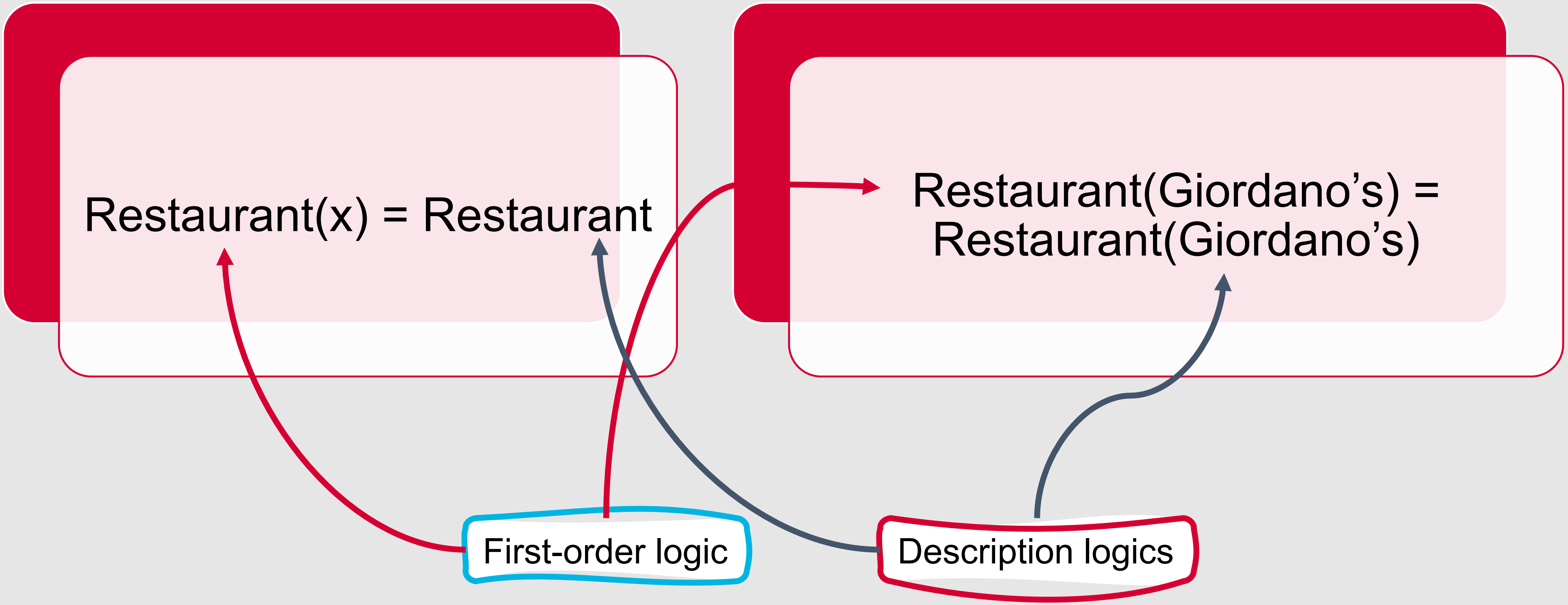
Description Logics

- How to add increased structure to semantics defined by models?
  - **Description Logics:** Different logical approaches that correspond to subsets of first-order logic
- More specific constraints make it possible to model more specific *forms* of inference

Description Logics

- Represent knowledge about:
  - Categories
  - Individuals who belong to those categories
  - Relationships that can hold among those individuals
- **Terminology:** The set of categories comprising a given application domain
- **Ontology:** Hierarchical representation of subset/superset relations among categories

Representation



Restaurant(x) = Restaurant

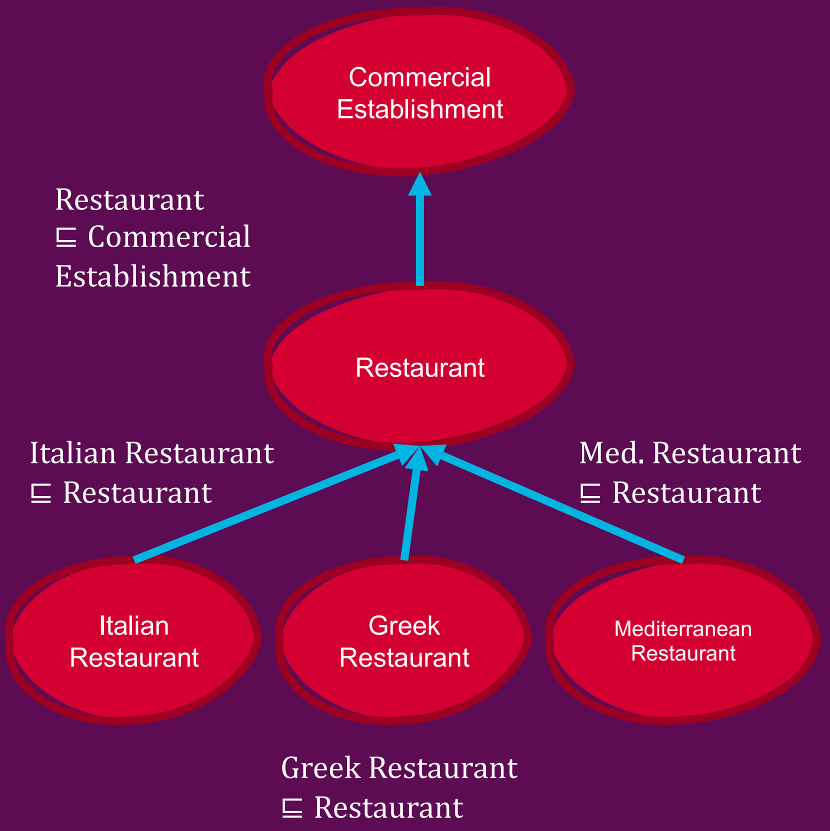
Restaurant(Giordano’s) = Restaurant(Giordano’s)

First-order logic

Description logics

Hierarchical Structure

- Can be directly specified using subsumption relations between concepts
  - **Subsumption:** All members of category *C* are also members of category *D*, or $C \sqsubseteq D$



Commercial Establishment
Restaurant ⊑ Commercial Establishment
Restaurant
Italian Restaurant ⊑ Restaurant
Med. Restaurant ⊑ Restaurant
Italian Restaurant
Greek Restaurant
Mediterranean Restaurant
Greek Restaurant ⊑ Restaurant

Category Membership

- Coverage or disjointness can be further specified using logical operators
  - $\text{Italian Restaurant} \sqsubseteq \textbf{NOT } \text{Greek Restaurant}$
  - $\text{Restaurant} \sqsubseteq$
    $\textbf{OR } (\text{Italian Restaurant, Greek Restaurant, Mediterranean Restaurant})$

Category Membership

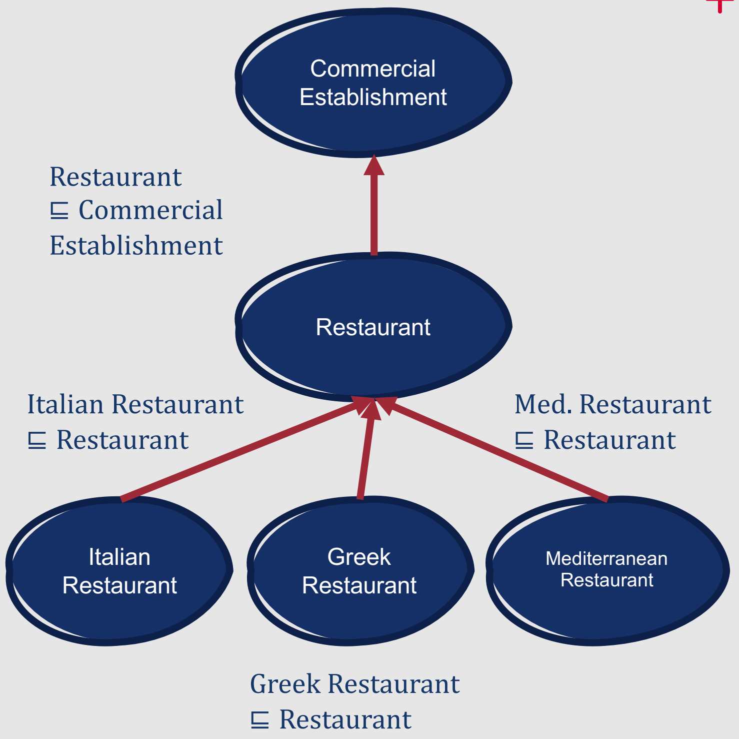
- Relations provide further information about category membership
  - $\text{Italian Cuisine} \sqsubseteq \text{Cuisine}$
  - $\text{Italian Restaurant} \sqsubseteq \text{Restaurant} \sqcap \exists\text{hasCuisine.ItalianCuisine} = \forall x \text{ItalianRestaurant}(x) \longrightarrow \text{Restaurant}(x) \land (\exists y \text{Serves}(x, y) \land \text{ItalianCuisine}(y))$

Hierarchical Structure

- Relations also allow us to explicitly define necessary and sufficient conditions for categories
  - $\text{Italian Restaurant} \sqsubseteq \text{Restaurant} \sqcap \exists\text{hasCuisine.ItalianCuisine}$
  - $\text{Greek Restaurant} \sqsubseteq \text{Restaurant} \sqcap \exists\text{hasCuisine.GreekCuisine}$

Inference

- Subsumption as a form of inference
  - Based on the facts in our terminology, does a superset/subset relationship exist between two concepts?


Commercial Establishment
Restaurant ⊑ Commercial Establishment
Restaurant
Italian Restaurant ⊑ Restaurant
Med. Restaurant ⊑ Restaurant
Italian Restaurant
Greek Restaurant
Mediterranean Restaurant
Greek Restaurant ⊑ Restaurant

Real-World Example of Description Logics

- **Web Ontology Language (OWL)**
  - Formally specifies semantic categories of the internet through the creation and deployment of ontologies for application areas of interest
  - Built using a description logic similar to that described in the previous slides

Summary: First-Order Logic

- In **model-theoretic semantics**, the model serves as a formal construct representing a particular state of affairs in the world
- **First-order logic** maps linguistic input to world knowledge using logical rules
- Core components of a first-order logic model are:
  - **Objects**
  - **Properties**
  - **Relations**
- We can apply **truth-conditional logic** (**and**, **or**, and **not** operators) to sentences to determine whether they fit a given model based on their included terms
- First-order logic makes use of both **existential** and **universal** quantifiers
- Inferences can be drawn from first-order logic statements using **modus ponens**
- **Description logic** models semantic domains using subsets of first-order logic, restricting expressiveness such that it guarantees the tractability of certain kinds of inference