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
  - **Syntactic:** Focused on **sentence structure**
  - **Semantic:** Focused on **meaning**

How are dependency grammars different from CFGs?

- CFGs are used to automatically generate constituent-based representations
    - Noun phrases, verb phrases, etc.
- Dependency grammars ignore phrase-structure rules, and instead 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

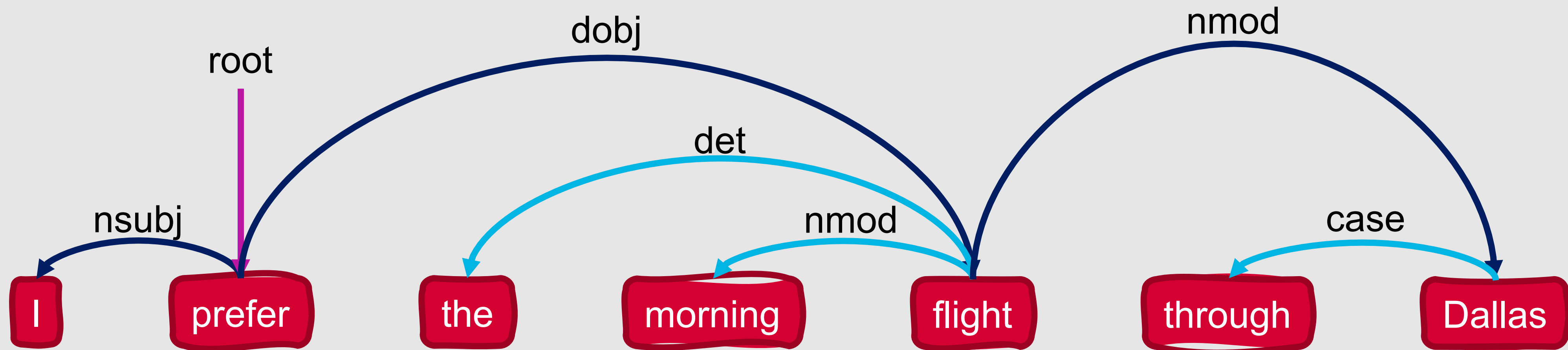
Dependency grammars can deal with languages that are morphologically rich and have a relatively free word order.

Morphologically rich: More inflections (changes to words that influence meaning or grammatical relation)

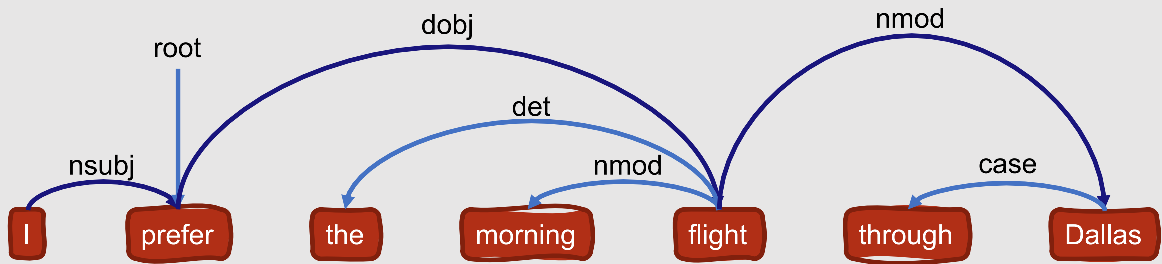
Free word order: Words can be moved around in a sentence but the overall meaning will remain the same (syntax is less important)

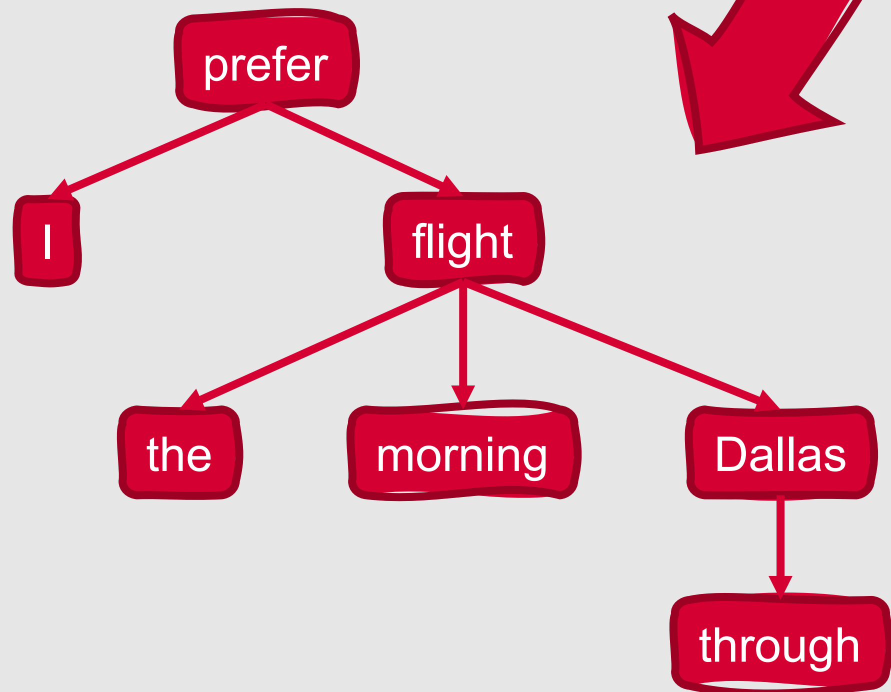
Typically, there is a trade-off between morphological richness and importance of syntax

Typed Dependency Structure

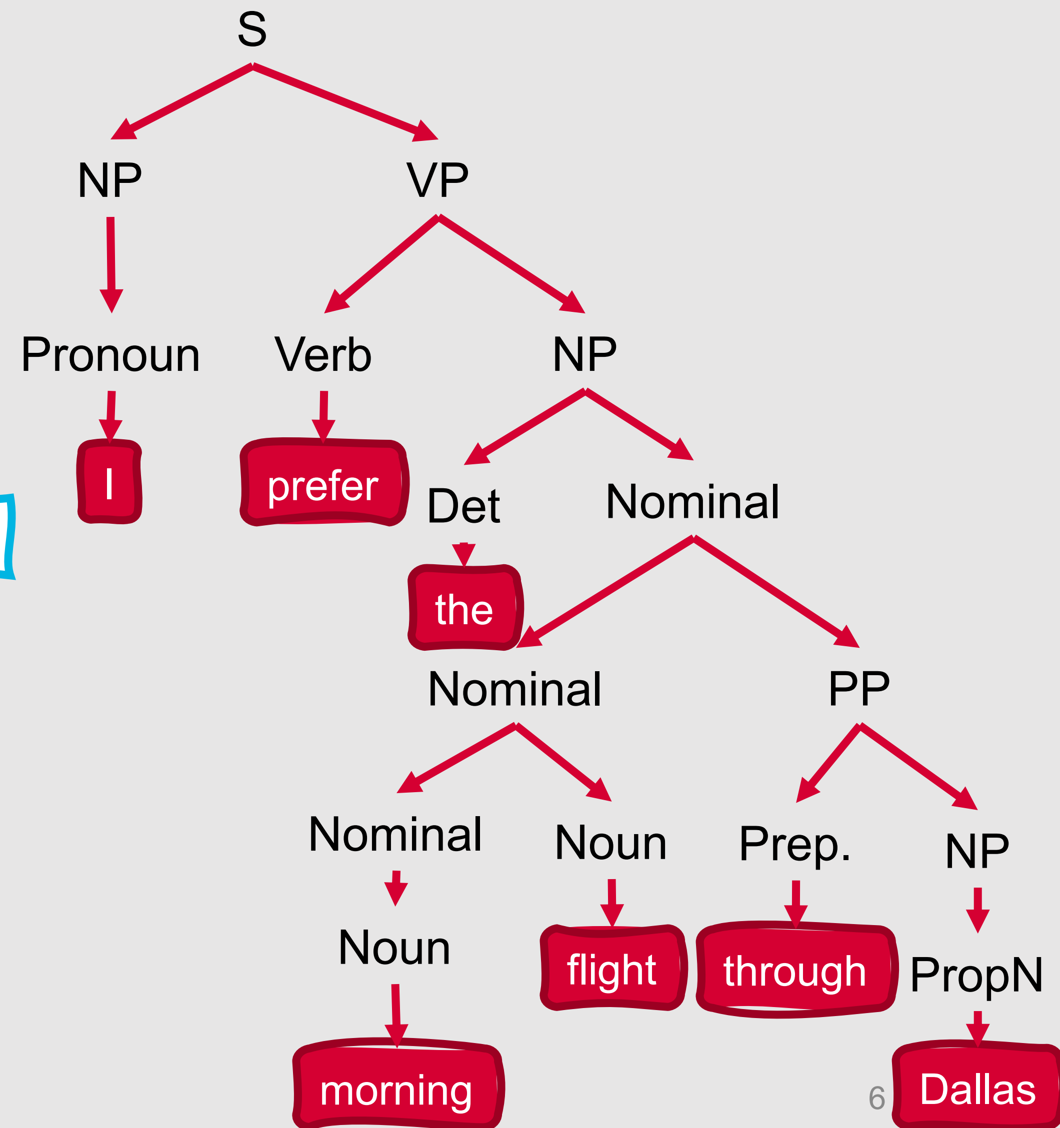


Comparison with Syntactic Parse





vs.



Why is dependency parsing useful?

- Dependency parsing provides an approximation of the semantic relationships between different words in a sentence and their arguments
- This information is useful for many NLP applications, including:
  - Coreference resolution
  - Question answering
  - Information extraction

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 more flexible languages (e.g., those with relatively free word order), the information encoded in these relation types often cannot be estimated from constituency trees

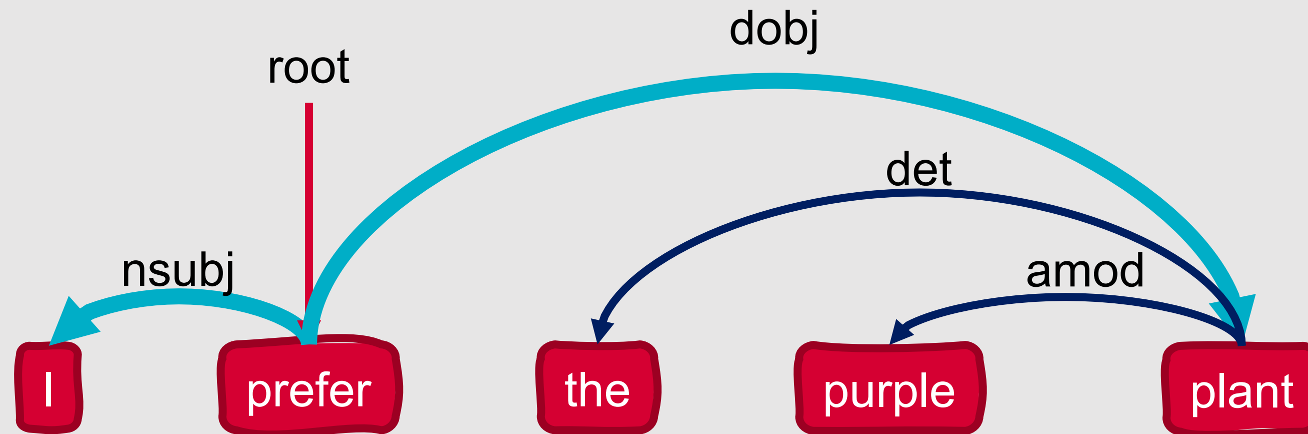
Just like with CFGs, there are a variety of taxonomies that can be used to label dependencies between words.

- These are often referred to as dependency treebanks
- A few of the most popular dependency treebanks include:
  - Stanford dependencies
  - CoNLL dependencies
  - Universal dependencies
- Just like with other corpora we've discussed so far, these treebanks are typically created by:
  - Having human annotators create dependency structures for a collection of sentences
  - Automatically creating initial dependency structures and then having human annotators manually correct those structures

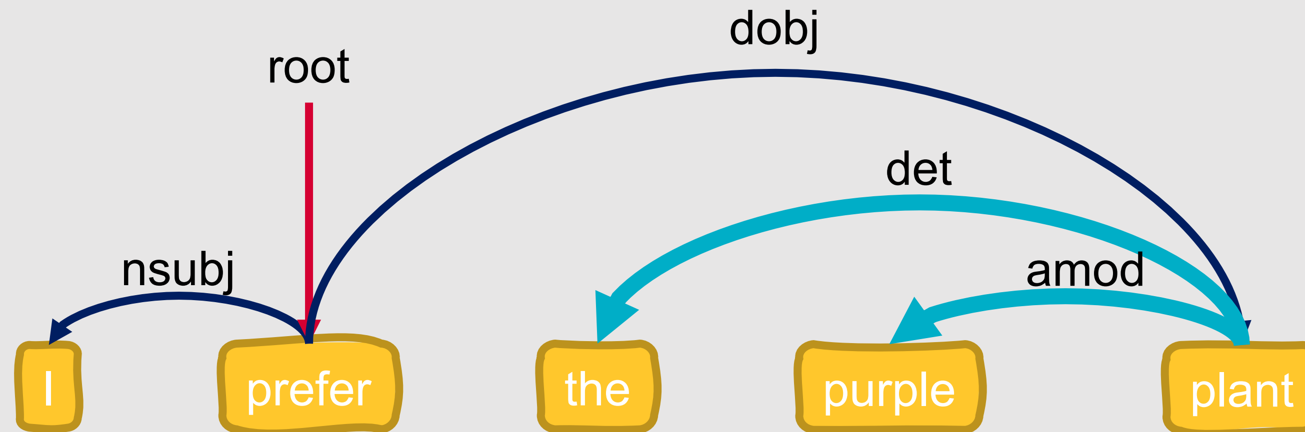
Recently, most researchers have moved toward using universal dependencies.

- Universal dependencies can be broken into two sets
  - **Clausal Relations:** Describe syntactic roles with respect to predicates (the part(s) of the sentence that say something about the subject)
  - **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 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 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 ase

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 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 (columns); Functional categories w.r.t. head (rows)

	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		aux
Dependents of Nominals	nmod appos nummod	acl		

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 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

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		
Dependents of Nominals	nmod appos nummod	acl	amod	det 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 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 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				
Non-Core Dependents of Clausal Predicates	obl vocative expl dislocated			aux cop mark
Dependents of Nominals				det 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 vocative expl dislocat	advcl	advmod [illegible]urse	aux cop mark
Dependents of Nominals	nmod appos nummod	acl	amod	det case

That was the goal.
det(goal, the)

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 complex languages, may include stems and affixes
- Arcs capture the grammatical relationships between those words

According to most grammatical theories, dependency structures:

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

- 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:

- Identify all head-dependent relations in the constituent tree
- Identify the correct dependency relations for those relations

One algorithm for doing this:

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

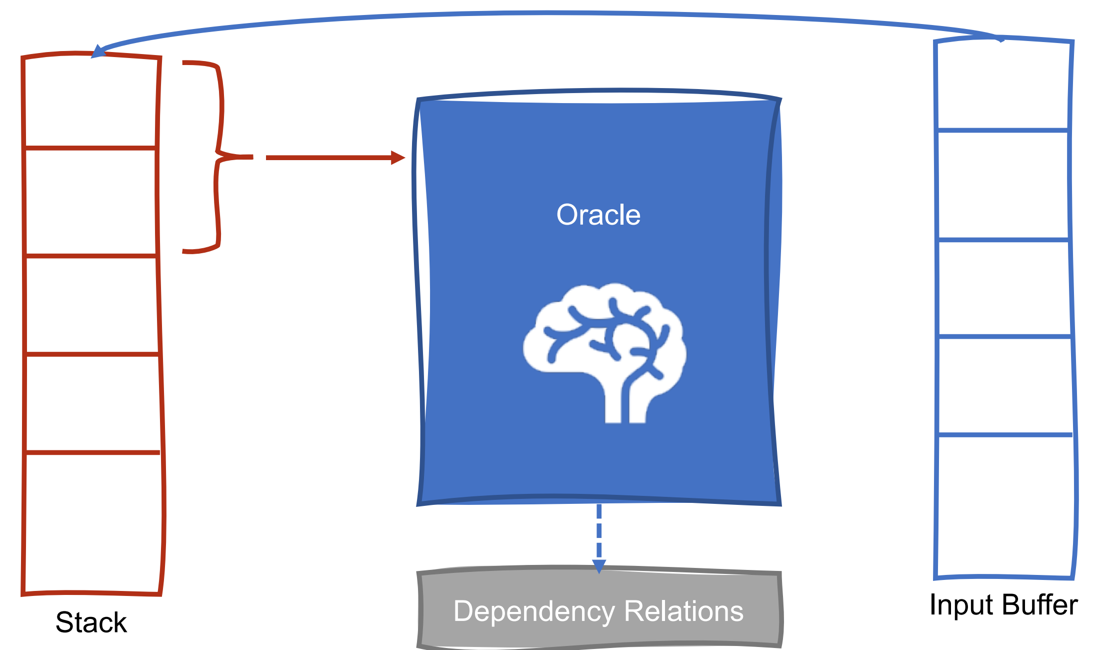
- Most noun phrases do not have much (or any) internal structure
- Morphological information has little to no presence in phrase structure trees
- For **low resource languages** in particular, most dependency treebanks are developed manually by human annotators rather than attempting to automatically translate from constituent to dependency parse

Types of Dependency Parsers

Transition	**Transition-based** • Build a single tree in a left-to-right (assuming a left-to-right language) 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

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 a set of **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
- Word list is initialized with all words in the sentence, in order
- Empty set of relations represents the parse

Final configuration:

- Stack should be empty
- Word list should be empty
- Set of relations represents the parse

Operators

- The operators used in transition-based parsing then perform 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)
```

When does the process end?

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

Is this another example of dynamic programming?

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

Arc Standard: Example

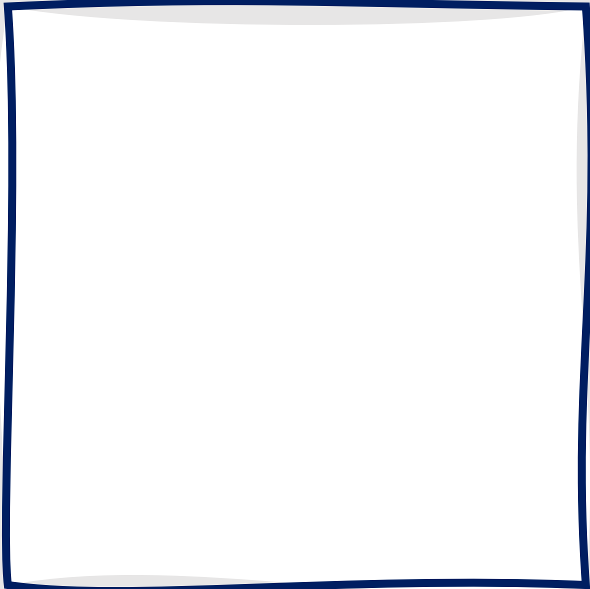
Input Buffer

book	me	the	morning	flight

Stack

root					

Relations



Arc Standard: Example

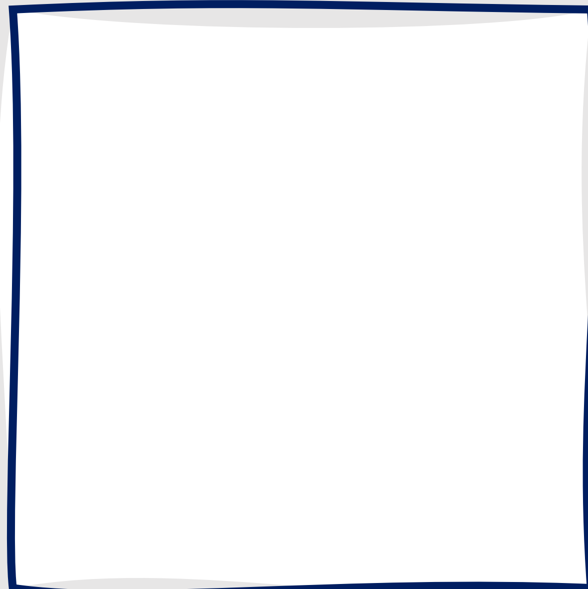
Input Buffer

	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

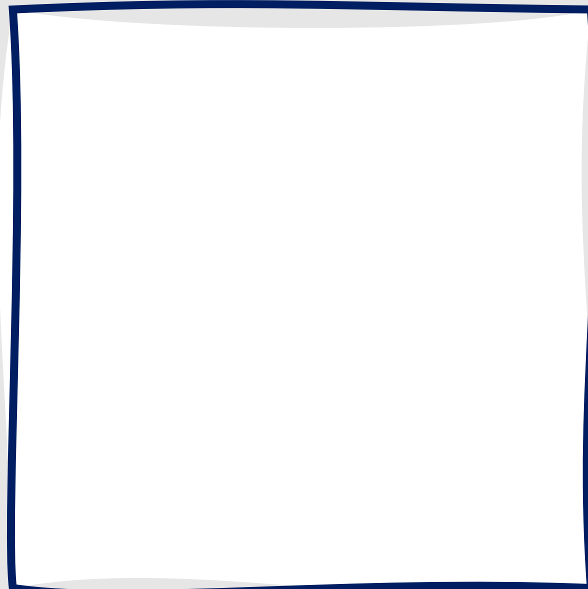
Input Buffer

		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

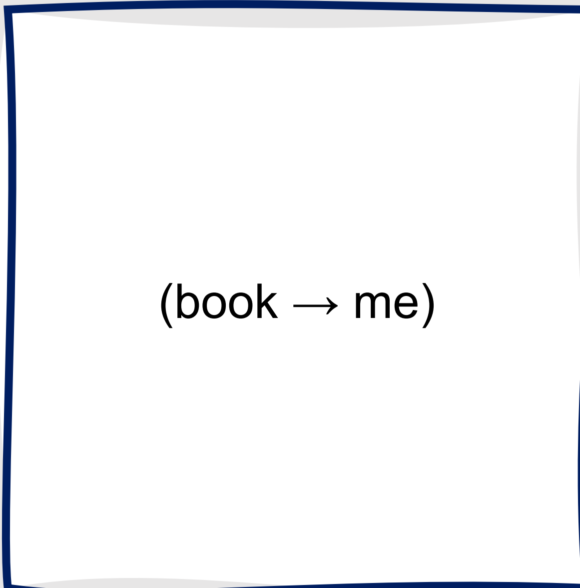
Add relation (book → me) to the set of relations

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: Shift, RightArc

Oracle selects Shift

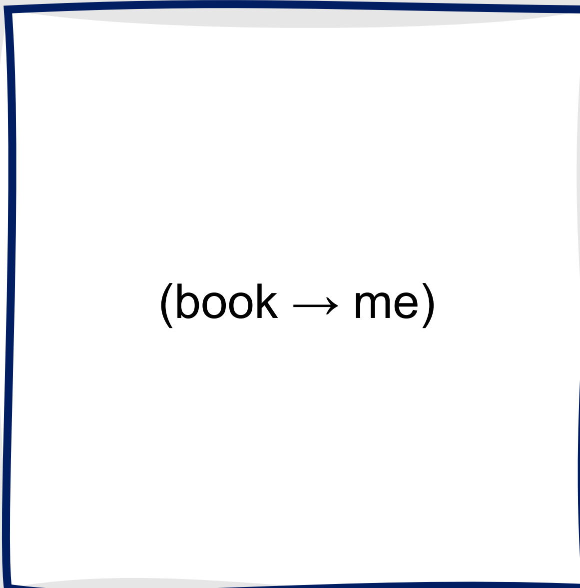
Shift **the** from the input buffer to the stack

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: Shift, RightArc, LeftArc

Oracle selects Shift

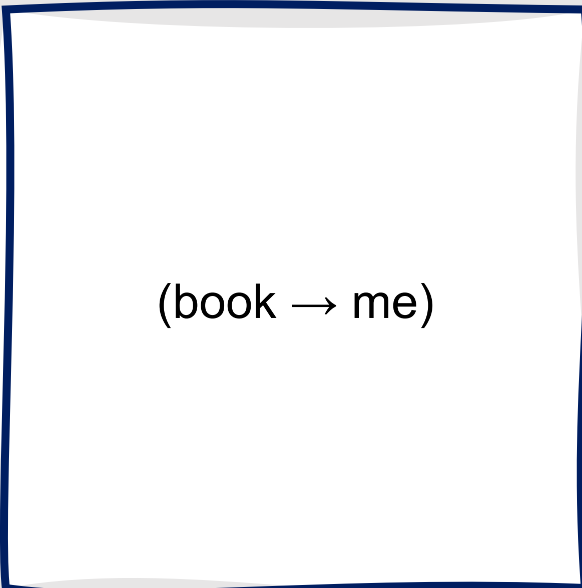
Shift **morning** from the input buffer to the stack

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: Shift, RightArc, LeftArc

Oracle selects Shift

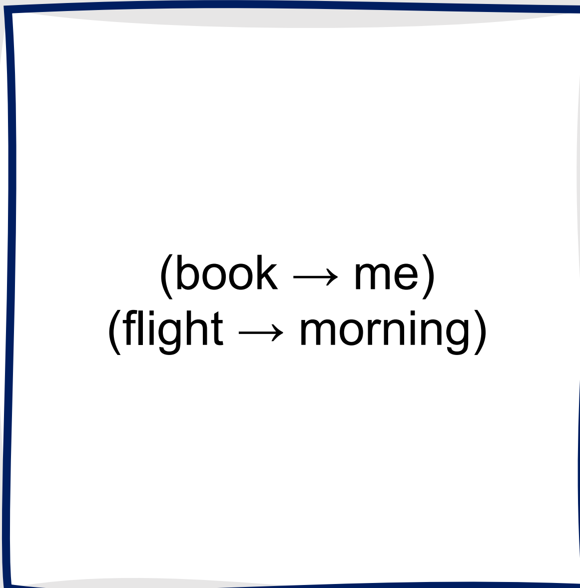
Shift **flight** from the input buffer to the stack

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: RightArc, LeftArc

Oracle selects LeftArc

Remove **morning** from the stack

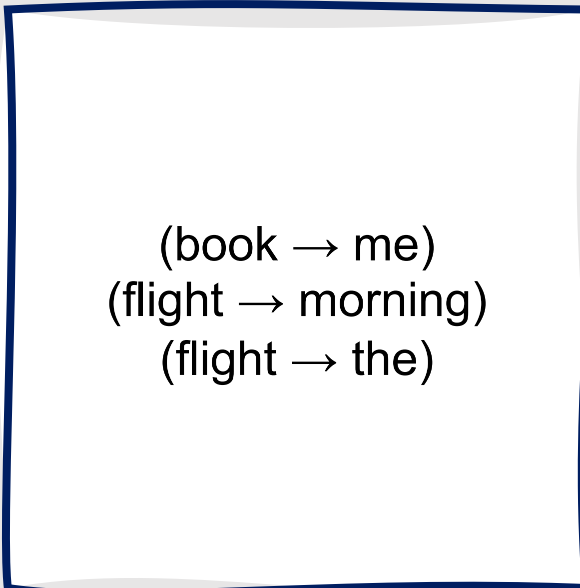
Add relation (flight → morning) to the set of relations

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: RightArc, LeftArc

Oracle selects LeftArc

Remove **the** from the stack

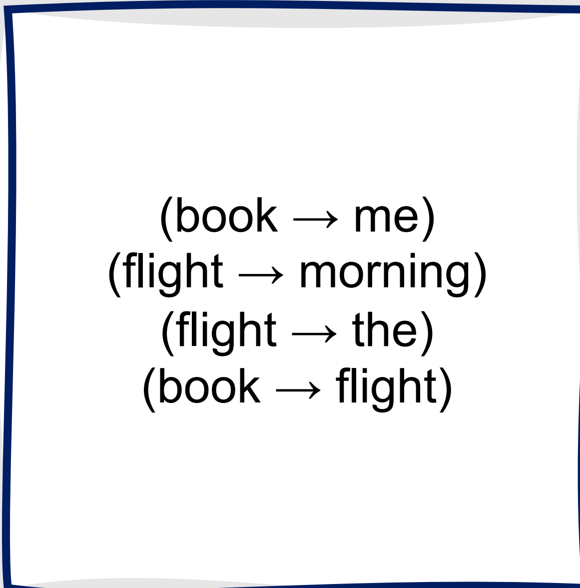
Add relation (flight → the) to the set of relations

Arc Standard: Example

Input Buffer

Stack

Relations



Valid options: RightArc, LeftArc

Oracle selects RightArc

Remove **flight** from the stack

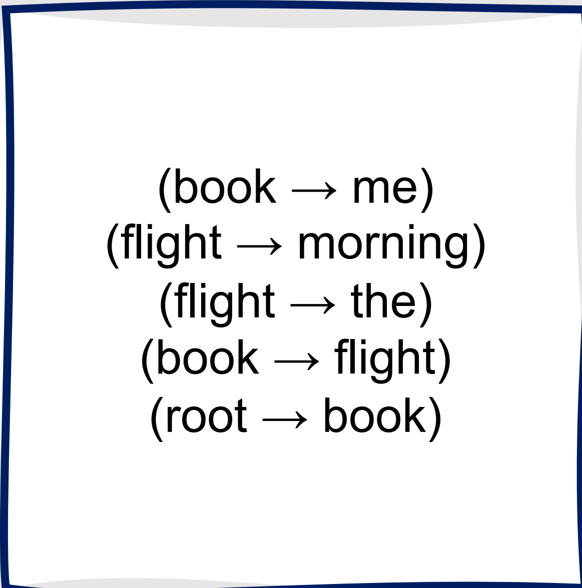
Add relation (book → flight) to the set of relations

Arc Standard: Example

Input Buffer

Stack

Relations



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

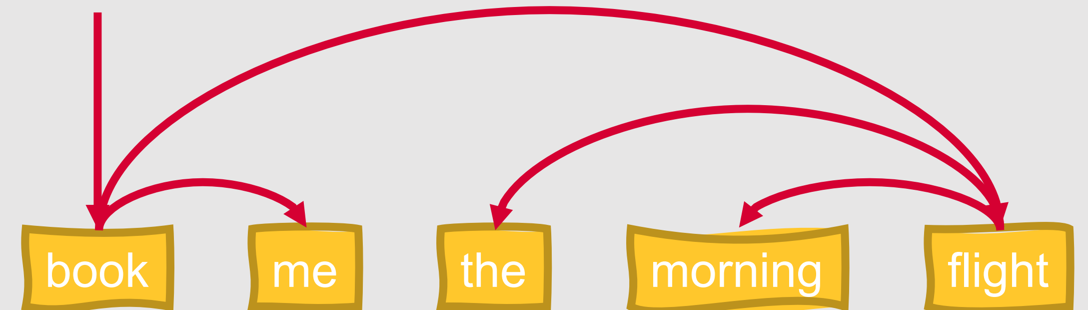
Stack

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?

- State of the art systems use **supervised machine learning** for this task
- This requires a training set of configurations labeled with correct transition operators
- The person designing the system needs to decide what types of features should be extracted from these configurations to best train the oracle (a machine learning model)
- The oracle will then learn which transitions to predict for previously-unseen configurations based on the extracted features and associated labels for configurations in the training set

What types of machine learning models are used as oracles?

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

Graph-based Dependency Parsing

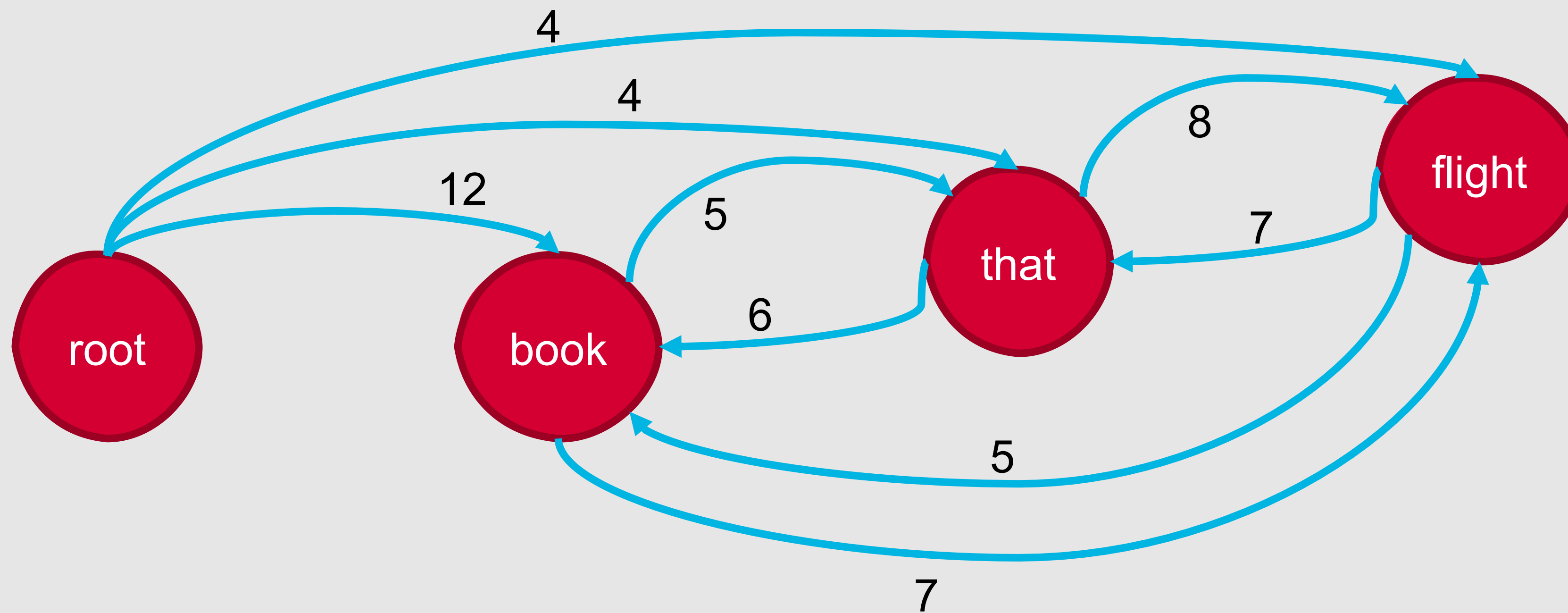
- Search through the space of possible trees for a given sentence, attempting to maximize some score
- This score is generally a function of the scores of individual subtrees within the overall tree
- **Edge-factored approaches** determine scores based on the scores of the edges that comprise the tree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

- Transition-based methods tend to have high accuracy on shorter dependency relations, but that accuracy declines as the distance between the two words increases
- This is largely due to the fact that transition-based methods are greedy …they can be fooled by seemingly-optimal local solutions
- Graph-based methods score entire trees, thereby avoiding that issue

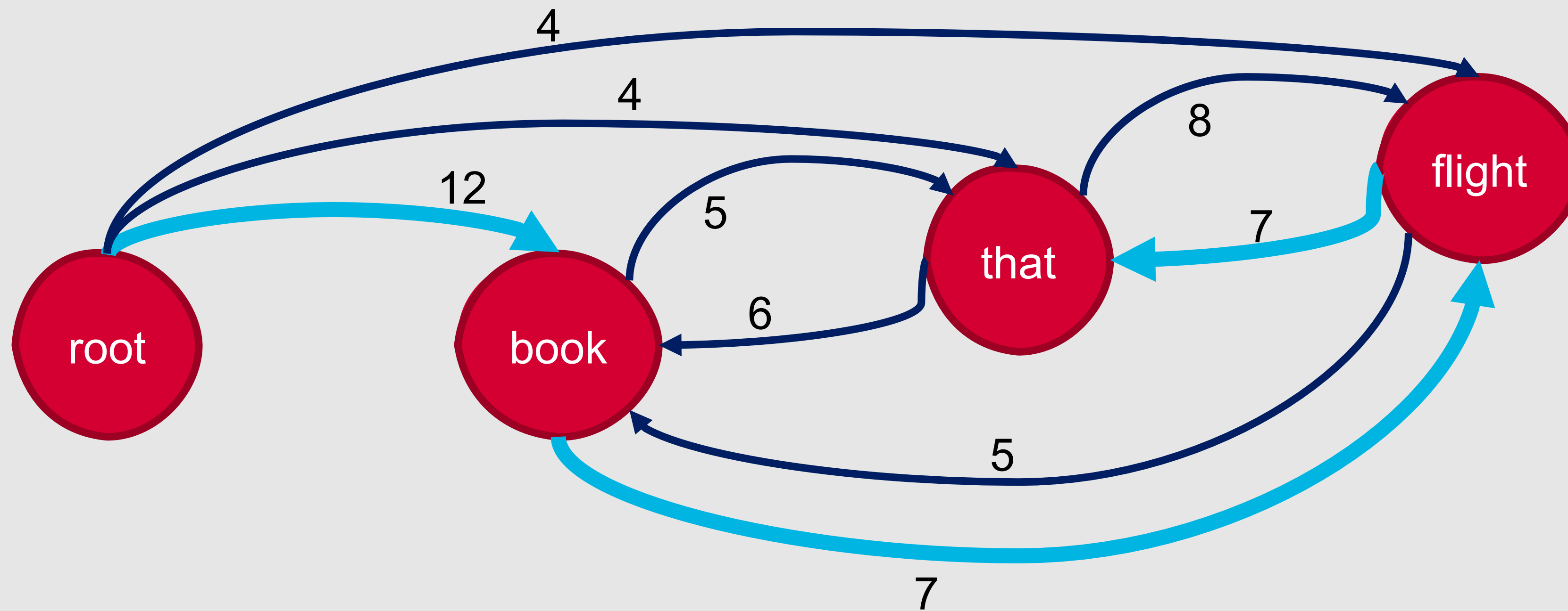
Maximum Spanning Tree

- 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

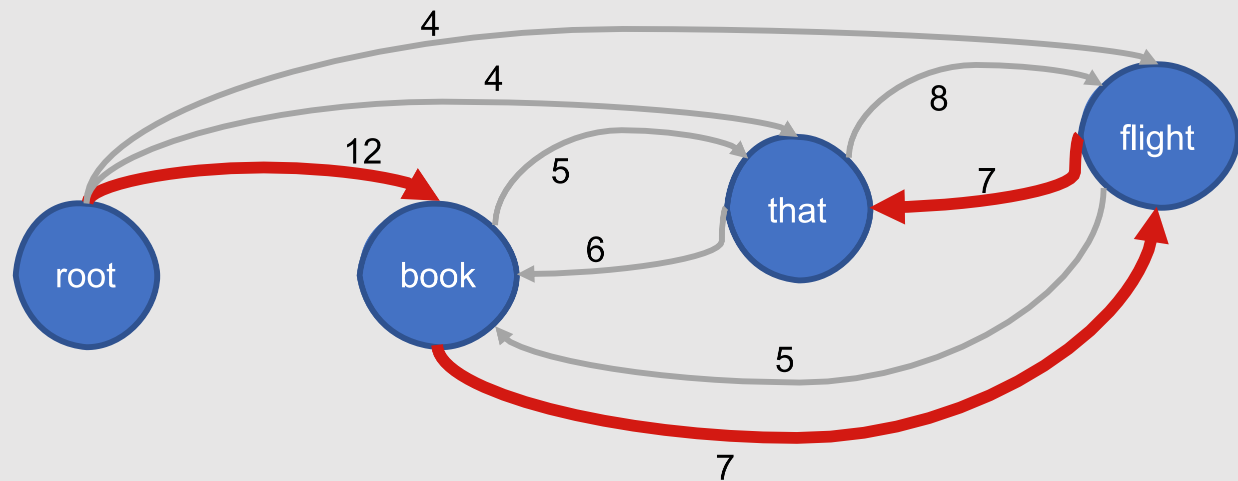


Maximum Spanning Tree: Example



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?

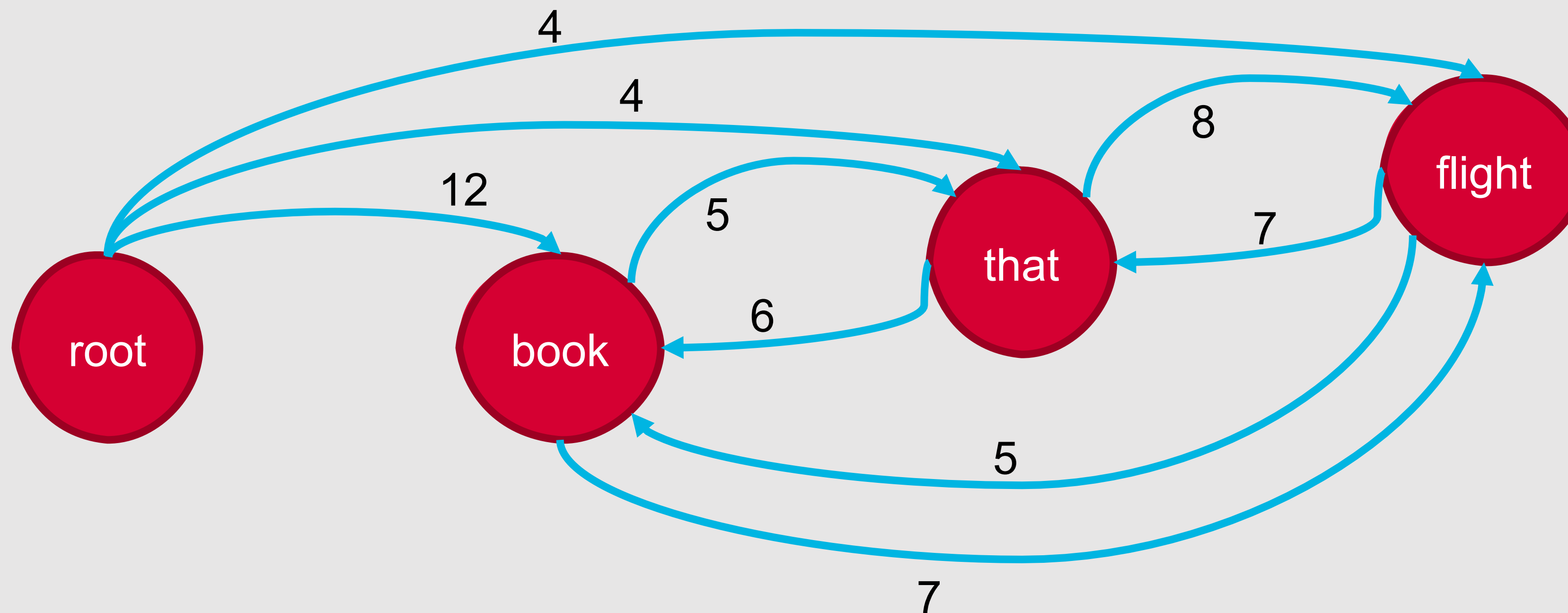
- **If the greedy selection process produces a spanning tree, then that tree is the maximum spanning tree**
- However, the greedy selection process may select edges that result in cycles
- If this happens, an alternate graph can be created that collapses cycles into new nodes, with edges that previously entered or exited the cycle now entering or exiting the new node
- The greedy selection process is then recursively applied to the new graph until a (maximum) spanning tree is found

Formal Algorithm

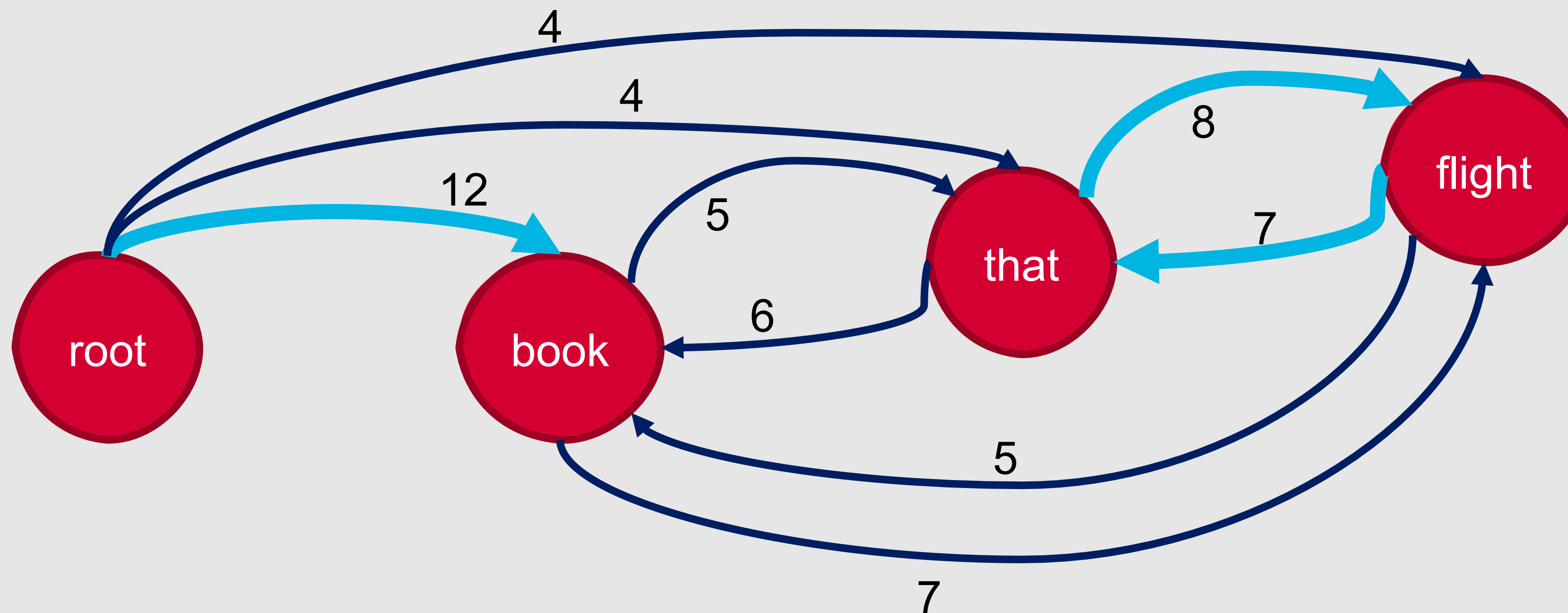
```
F ← []
T ← []
score’ ← []
for each v in V do:
        bestInEdge ← argmax_{e=(u,v)∈E} score[e]
        F ← F ∪ bestInEdge
        for each e = (u,v) ∈ E do:
                score’[e] ← score[e] - score[bestInEdge]

        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
```

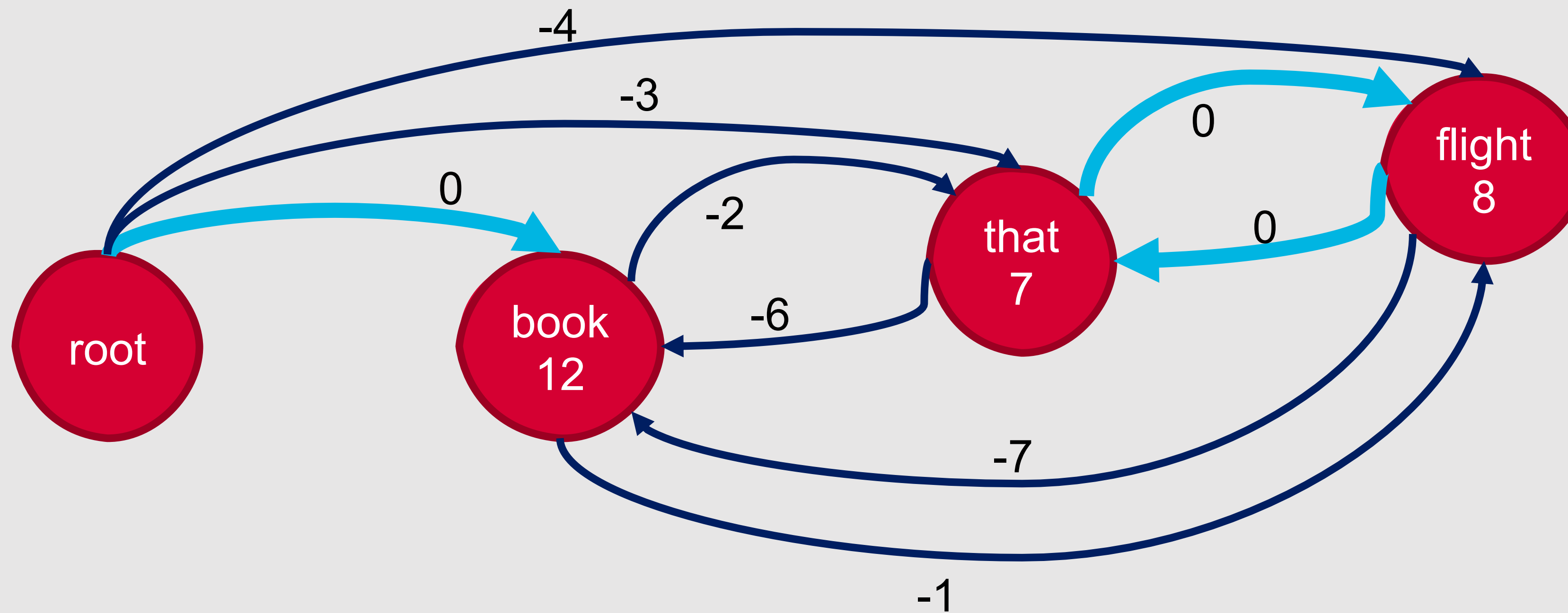
Maximum Spanning Tree: Updated Example



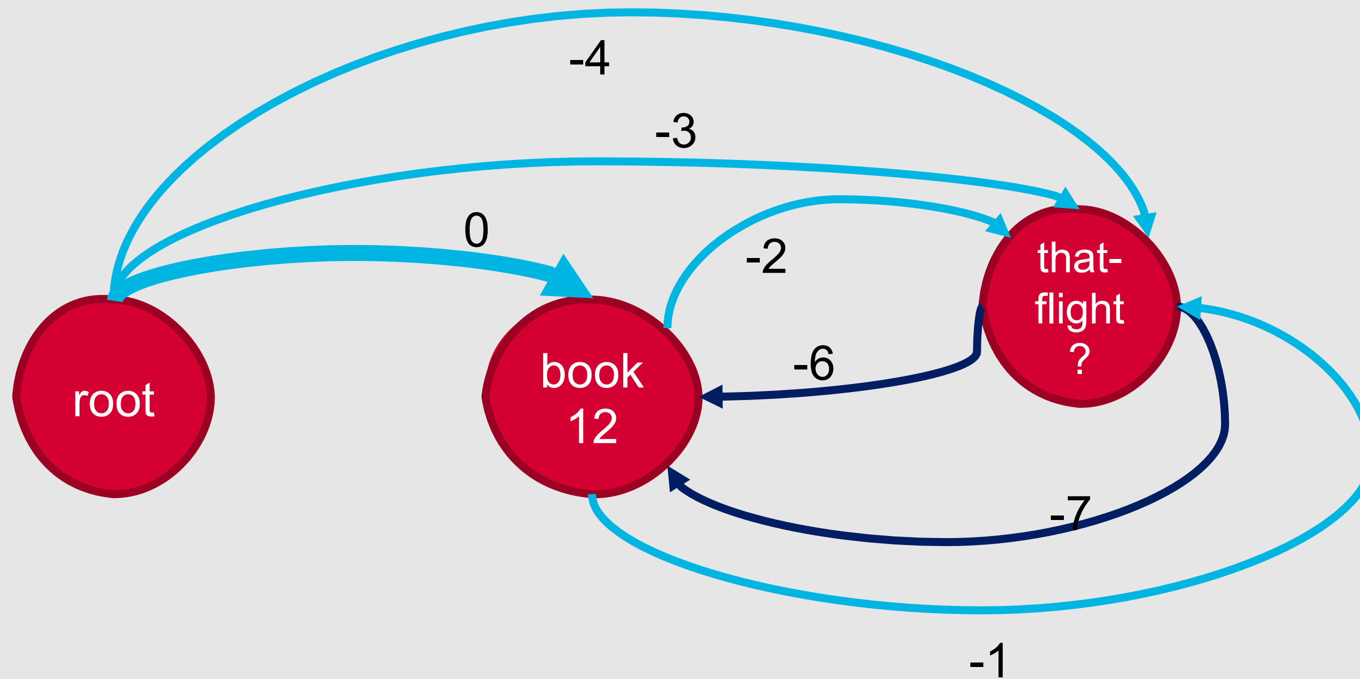
Maximum Spanning Tree: Updated Example



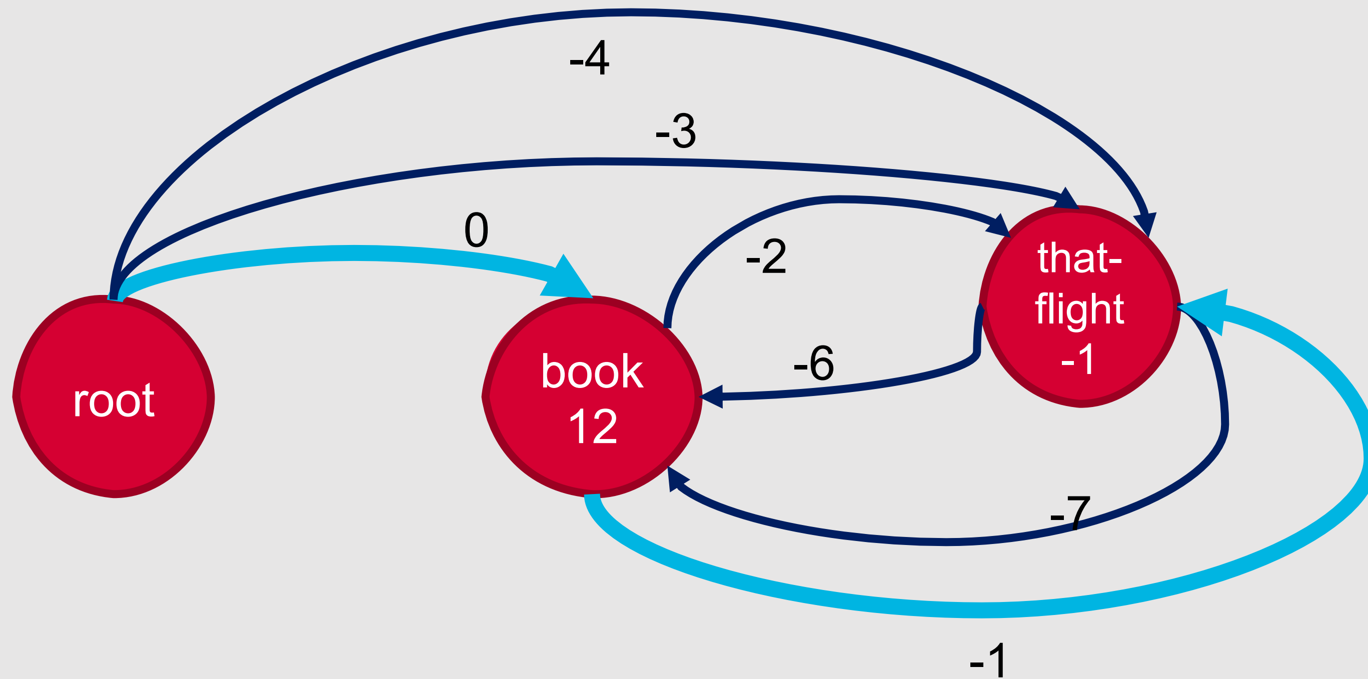
Maximum Spanning Tree: Updated Example



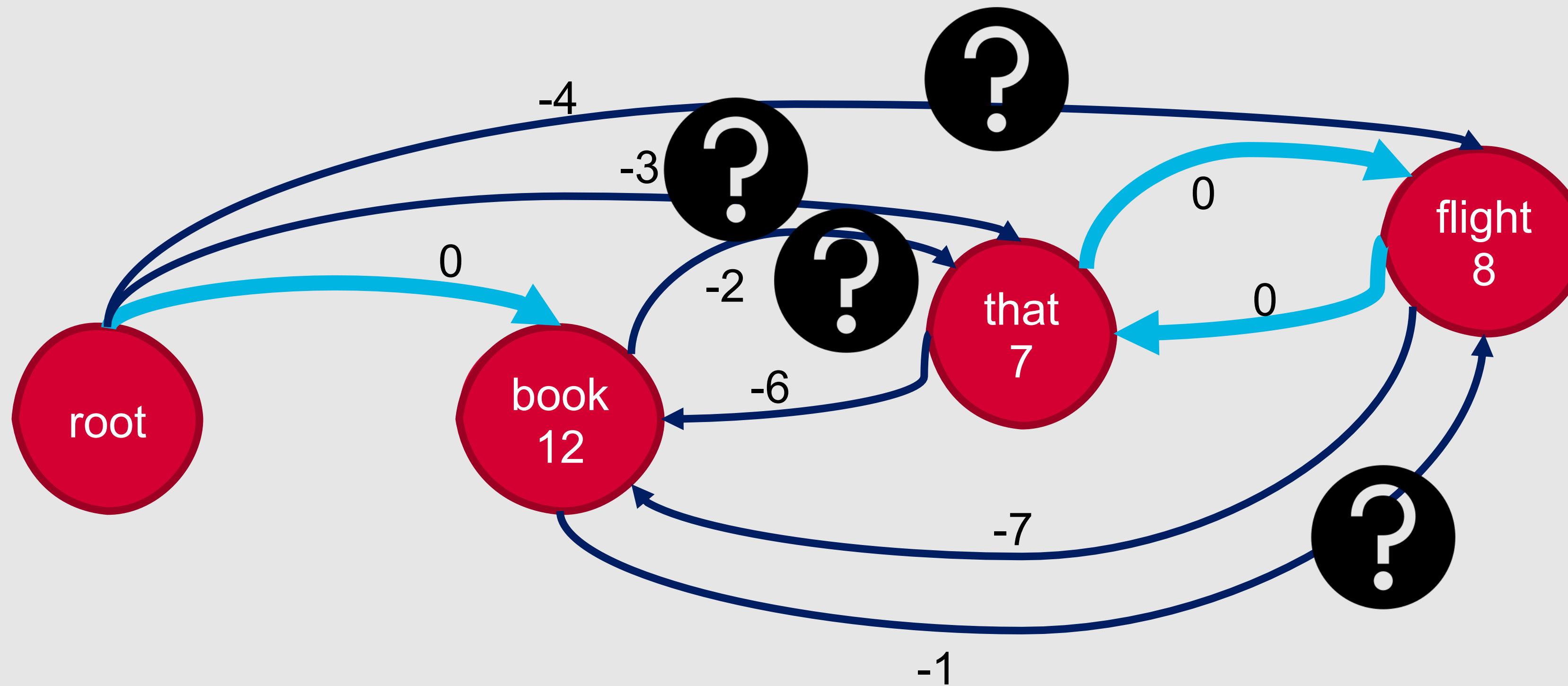
Maximum Spanning Tree: Updated Example



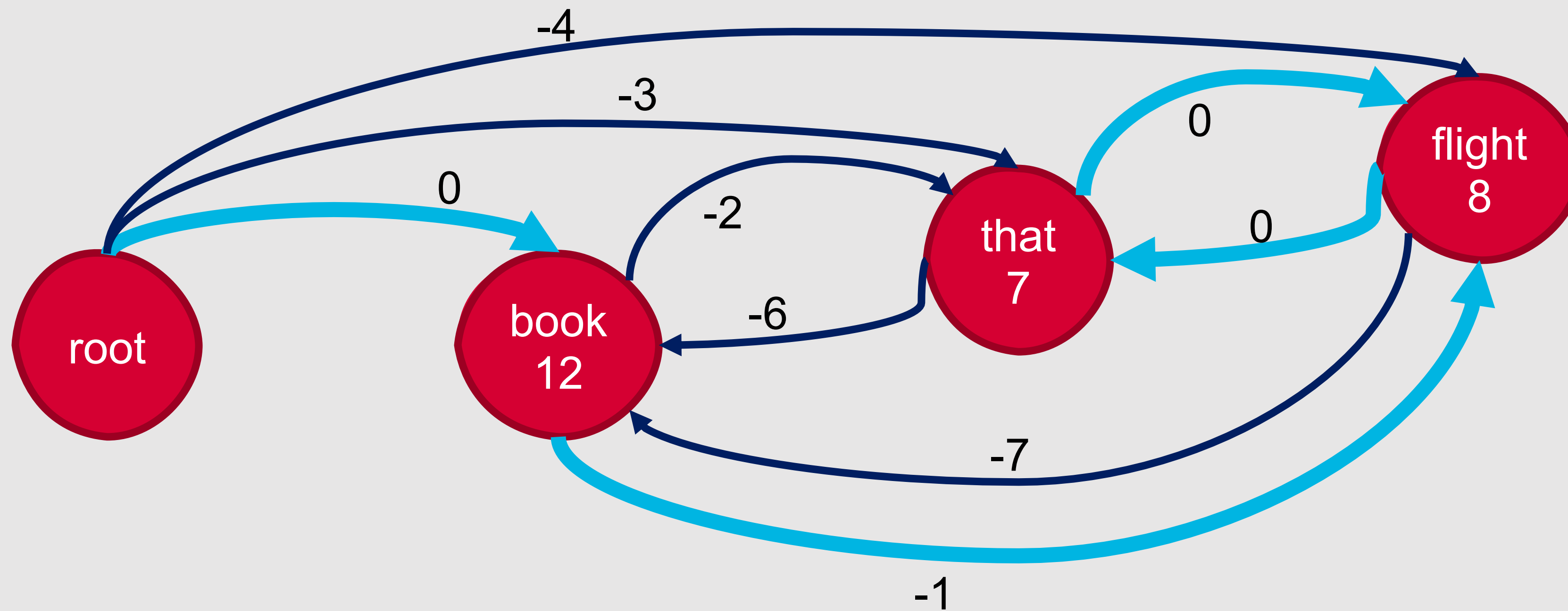
Maximum Spanning Tree: Updated Example



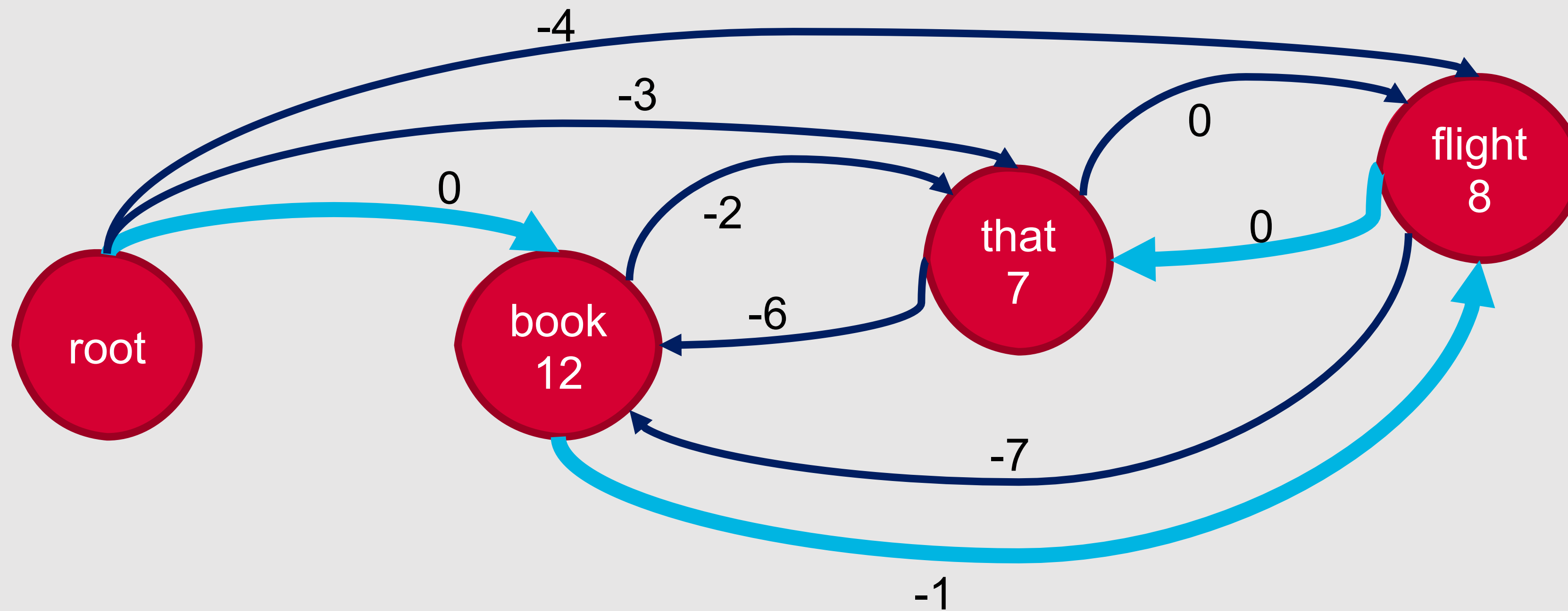
Maximum Spanning Tree: Updated Example



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
- Common features can include:
  - 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

Summary: Dependency Parsing

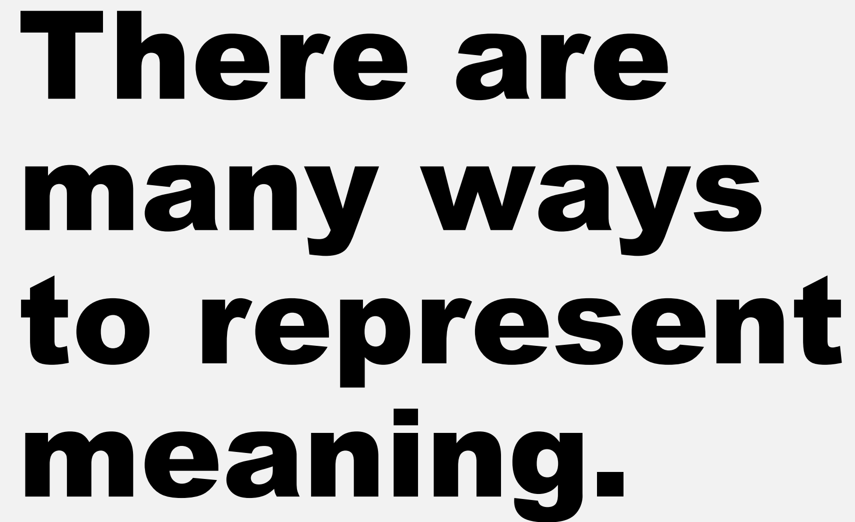
- **Dependency parsing** is the process of automatically determining **directed relationships between words** in a source sentence
- Many dependency taxonomies exist, but the most common taxonomy for English text is the set of **universal dependencies**
- Dependency parsers can be **transition-based** 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

Why do we need meaning representations?

- Somehow, we need to bridge the gap between **linguistic input** and non-linguistic **world knowledge** to perform semantic processing tasks
- Everyday examples of (human) semantic processing:
  - Answering essay questions on exams
  - Deciding what to order at a restaurant
  - Detecting sarcasm
  - Following recipes
  - Learning how to convert sentences to first-order logic

Meaning Representations

- **Goal: Represent commonsense world knowledge in logical form**
- These representations are created and assigned to linguistic inputs via **semantic analysis**



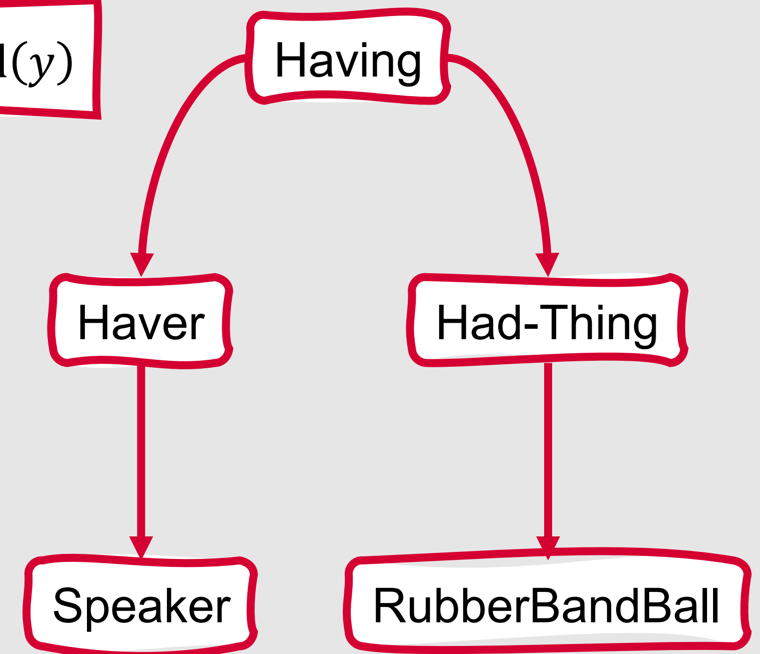
- First-Order Logic
- Semantic networks
- Conceptual dependencies
- Frame-based representations
- All of these approaches assume that meaning representations consist of **structures** composed from a set of **symbols**
  - **Symbols:** Representational vocabulary

Sample Meaning Representations

I have a rubber band ball.

$\exists x, y \text{ Having}(x) \wedge \text{ Haver}(x, Speaker) \wedge \text{HadThing}(x, y) \wedge \text{RubberBandBall}(y)$

Having
  Haver: Speaker
  HadThing: RubberBandBall



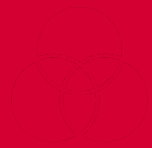
Symbols

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

Having	
Haver:	Speaker
HadThing:	RubberBandBall

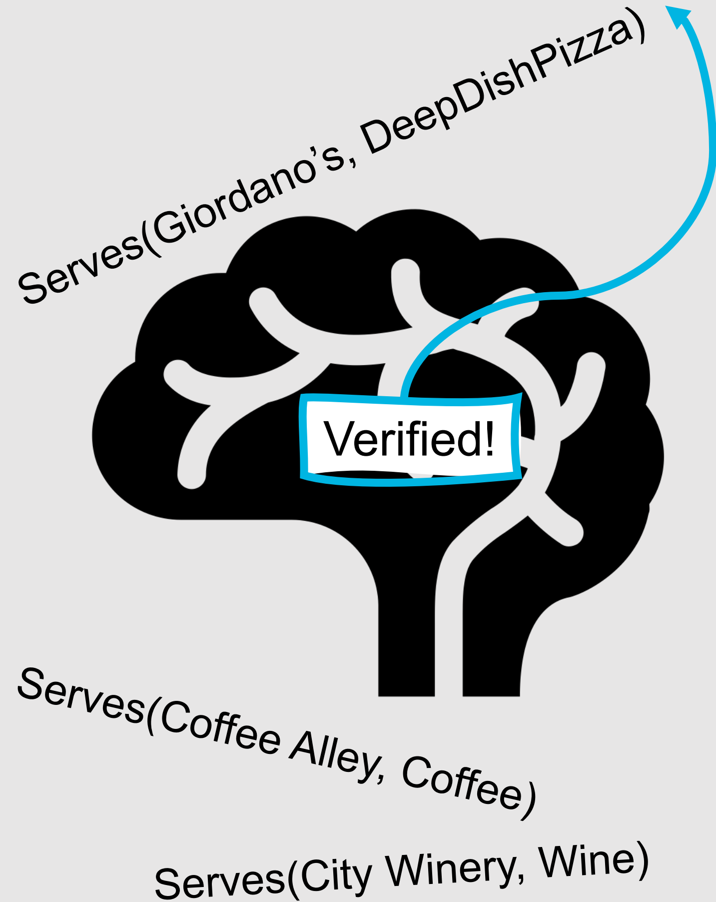
Basic Characteristics of Meaning Representations

- Verifiability
- Unambiguous Representations
- Canonical Form
- Inference and Variables
- Expressiveness



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

- 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
- Specifically, individual expressions can have different meaning representations depending upon the circumstances in which they occur

Unambiguous Representations

Let’s eat somewhere near SEO.

Let’s eat somewhere near SEO.

- We’ll ignore ambiguities arising from figurative language in this course, and focus on the semantic ambiguities that can still arise from literal expressions
- To resolve semantic ambiguities, computational methods must be employed to 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

Canonical Form

- Ambiguity means that a given sentence could 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(Coffee Alley, 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

Model-Theoretic Semantics

What do most meaning representation schemes share in common?

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

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

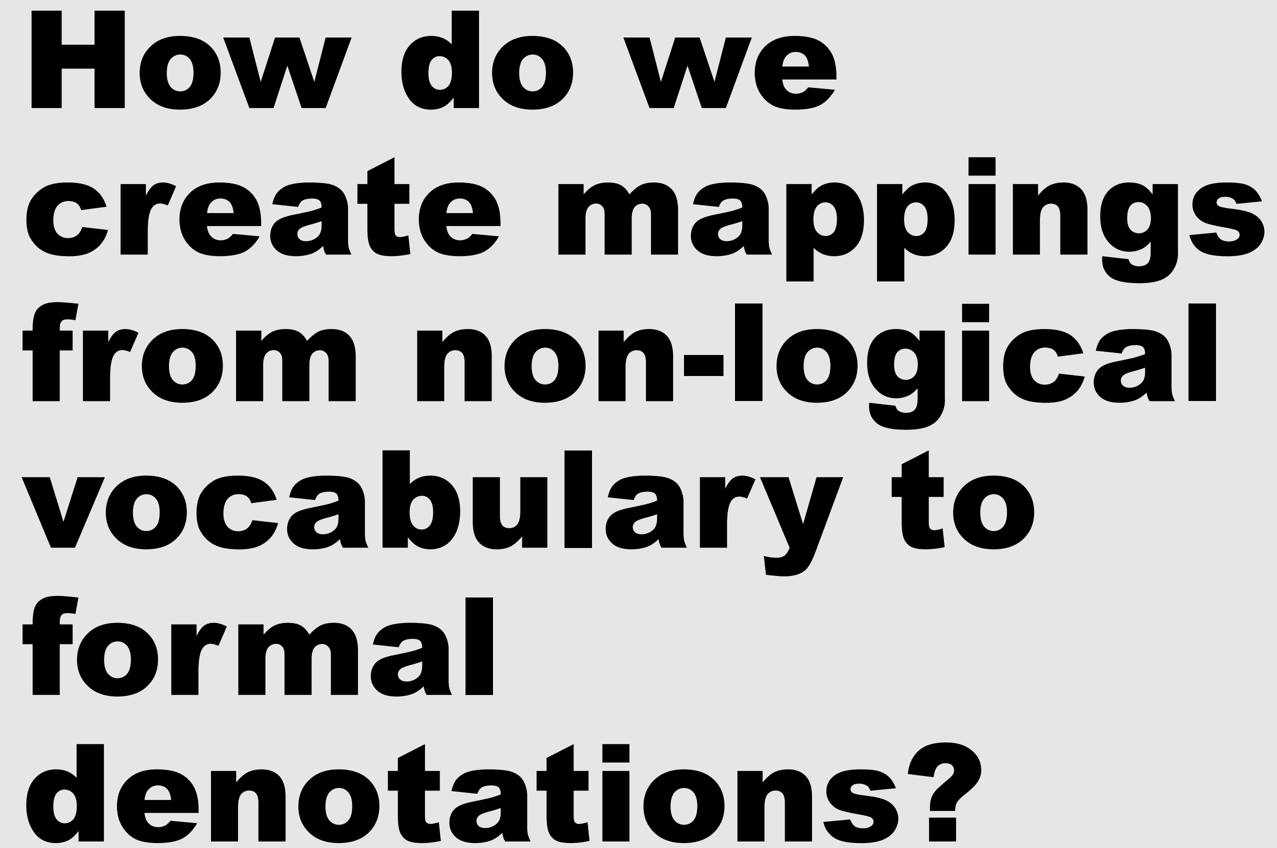
Expressions (words or phrases) in the meaning representation language can be mapped to elements of the model

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, links, etc. 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, Mina, Mohammad
- **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
  - StudentIn = {(CS421, Devika), (CS421, Guiseppe)}



We create functions (interpretations)!

Example Application

- 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*, *Usman*, *Nikolaos*, and *Mina*
- We'll use the constants *a*, *b*, *c*, and *d* to refer to those respective elements

Example Application

patron = {Natalie, Usman, 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, Usman, 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, Usman, 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, Usman, 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, Usman, 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 Usman likes Artopolis.
  - Mina likes fast restaurants.
  - Not everybody likes IDOF.

Example Application

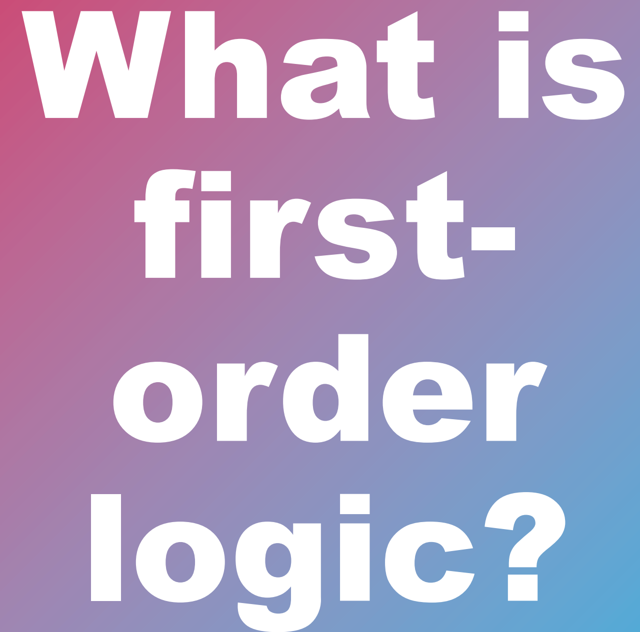
patron = {Natalie, Usman, Nikolaos, Mina} = {a, b, c, d}

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



- 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

- Terms:
  - **Constants:** Specific objects in the world being described
    - Conventionally depicted as single capitalized letters (A, B) or words (Natalie, Usman)
    - 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 $\wedge$
  - or $\vee$
  - implies $\rightarrow$
- They can also be **negated**
  - not $\neg$

Variables and Quantifiers

- Two basic operators in first-order logic are:
  - $\exists$: The existential quantifier
    - Pronounced “there exists”
  - $\forall$: The universal quantifier
    - Pronounced “for all”
- These two operators make it possible to represent many more sentences!
  - a restaurant → $\exists x$ Restaurant(x)
  - all restaurants → $\forall x$ 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 Usman likes Artopolis.
  - Like(Nikolaos, Giordano’s) $\wedge$ Like(Usman, 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
- The model-theoretic approach defines meaning based on truth-conditional mappings between expressions in a meaning representation and the state of affairs being modeled

We can determine truth based on the presence of specified terms and predicates.

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, Usman, 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 Usman likes Giordano's.

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

patron = {Natalie, Usman, 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 Usman likes Giordano's.

Likes(Natalie, Giordano's) ∧ Likes(Usman, Giordano's)

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

patron = {Natalie, Usman, Shahla, Yatri} = {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 Usman likes Giordano’s.

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

Likes(a, e) ∧ Likes(b, e)

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

patron = {Natalie, Usman, Shahla, Yatri} = {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 Usman likes Giordano's.

Likes(Natalie, Giordano's) ∧ Likes(Usman, Giordano's)

Likes(a, e) ∧ Likes(b, e)

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

patron = {Natalie, Usman, Shahla, Yatri} = {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 Usman likes Giordano's.

Likes(Natalie, Giordano's) ∧ Likes(Usman, Giordano's)

Likes(a, e) ∧ Likes(b, e)

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

patron = {Natalie, Usman, Shahla, Yatri} = {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 Usman likes Giordano's.

Likes(Natalie, Giordano's) ∧ Likes(Usman, 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:

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

Example: Inference

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

conditional statement accepted ✓

Example: Inference

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

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

conditional statement accepted ✓

Example: Inference

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

$$\frac{\mathrm{GreekRestaurant}(Artopolis) \quad \forall x\ \mathrm{GreekRestaurant}(x) \Rightarrow \mathrm{Serves}(x, GreekFood)}{\mathrm{Serves}(Artopolis, GreekFood)}$$

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
  - $\exists e\ Drink(Natalie, e)$
- If we determine that the actor must drink something specific: $\exists e\ Drink(Natalie, e) \wedge Beverage(e, tea)$
- More generally, we could define the representation:
  - $\exists e\ Drink(e) \wedge Drinker(e, Natalie) \wedge Beverage(e, tea)$
- 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 Shahla 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.

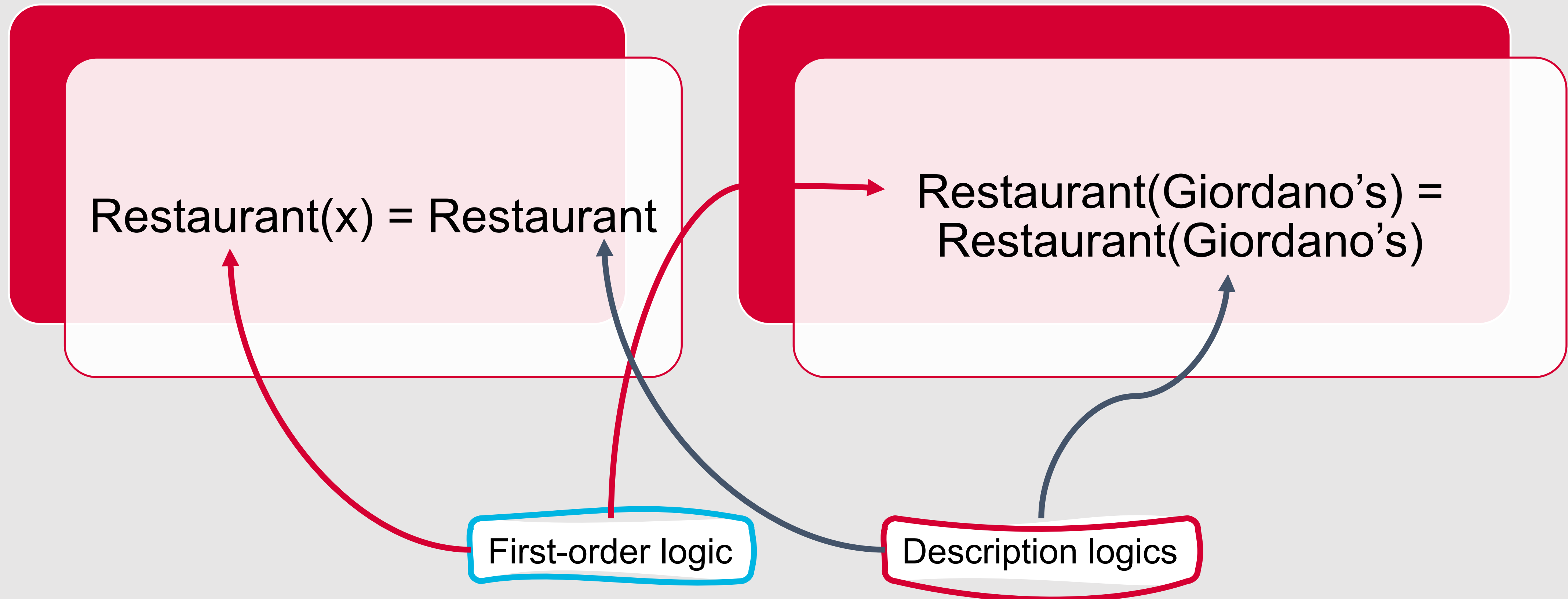
Description Logics

- How to add increased structure to semantics defined by models so far?
  - **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

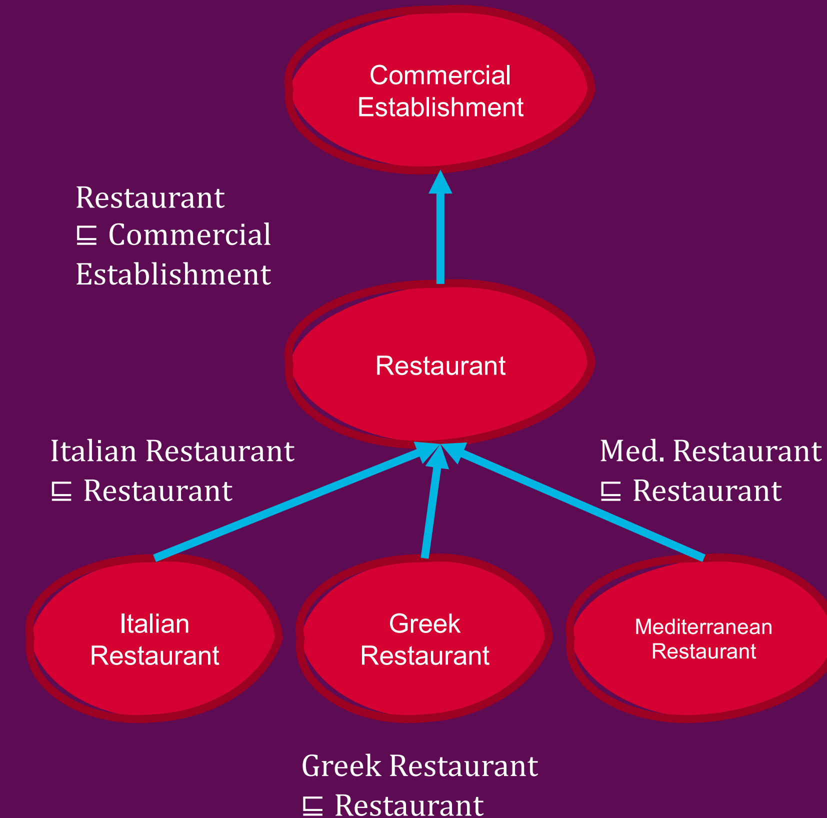
- Represent knowledge about:
  - Categories
  - Individuals who belong to those categories
  - Relationships that can hold among those individuals
- **Terminology:** The set of categories comprising a given application domain
- **TBox:** The portion of the knowledge base containing the terminology
-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

Representation



Hierarchical Structure

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



Category Membership

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

Category Membership

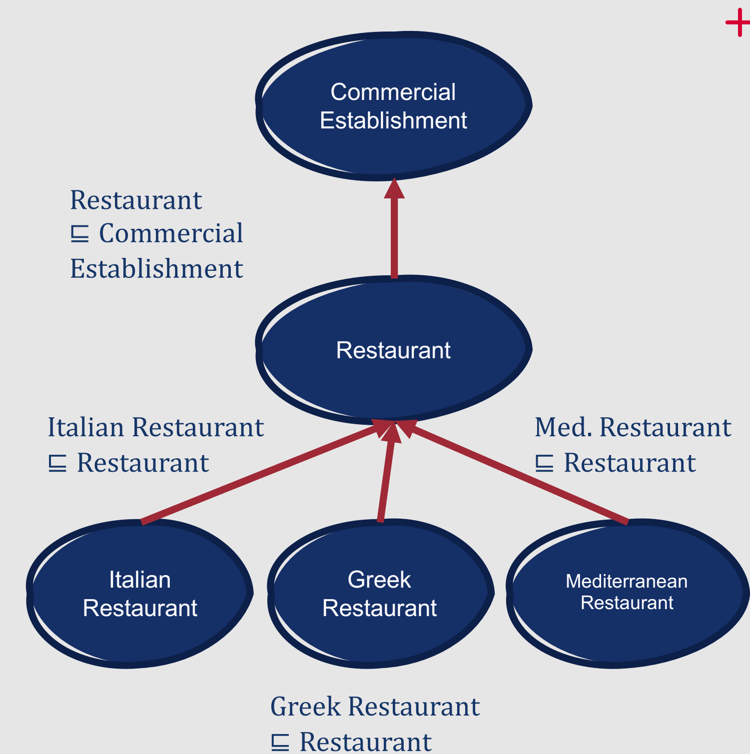
- Relations provide further information about category membership
  - $\text{Italian Cuisine} \sqsubseteq \text{Cuisine}$
  - $\text{Italian Restaurant} \sqsubseteq \text{Restaurant} \sqcap \exists\text{hasCuisine.ItalianCuisine} = \forall x \text{ItalianRestaurant}(x) \longrightarrow \text{Restaurant}(x)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

Hierarchical Structure

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



- **First-order logic** is a way to represent meaning by mapping 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**