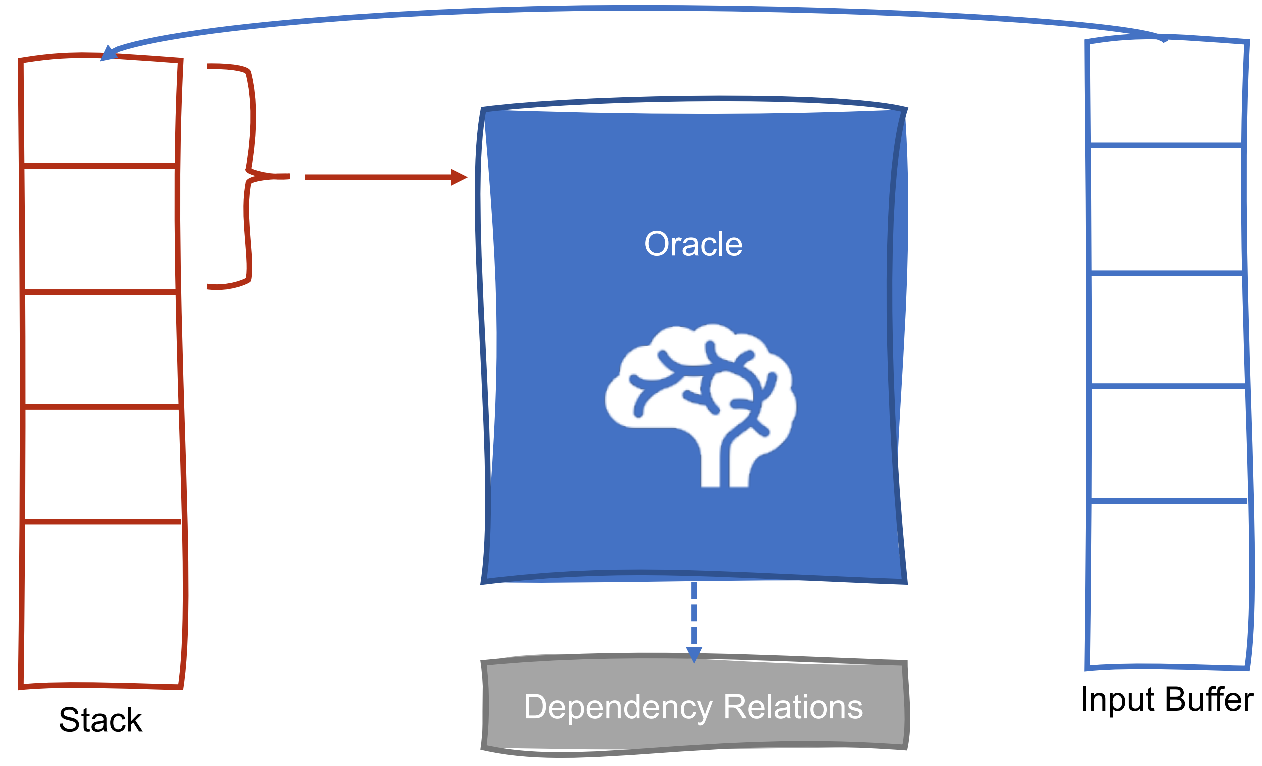
Transition-Based Dependency Parsing

Natalie Parde
UIC CS 421

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				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)

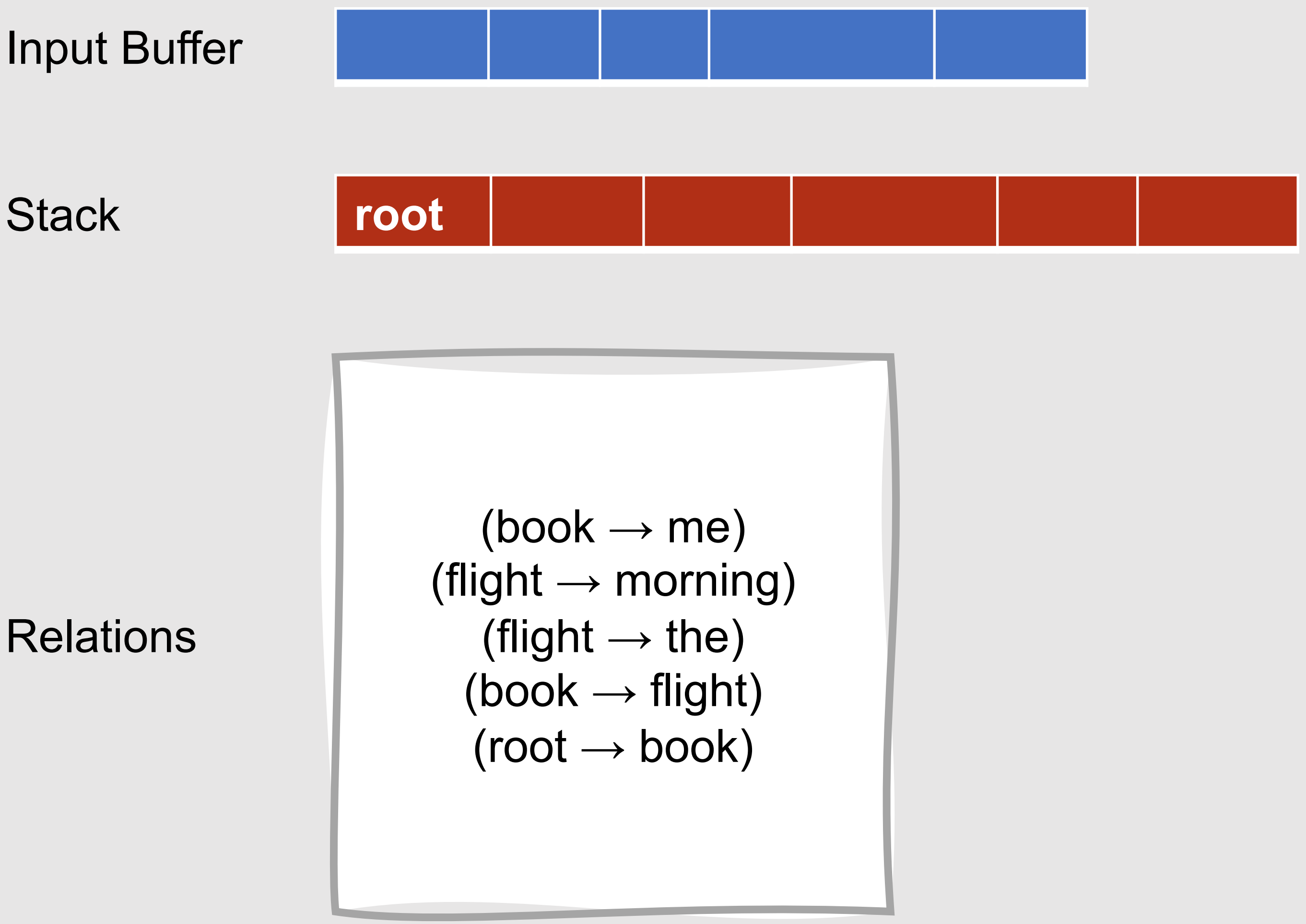
Valid options: RightArc, LeftArc

Oracle selects RightArc

Remove **flight** from the stack

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

Arc Standard: Example



Valid options: RightArc

Oracle selects RightArc

Remove **book** from the stack

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

Arc Standard: Example

Valid options: None

State is final

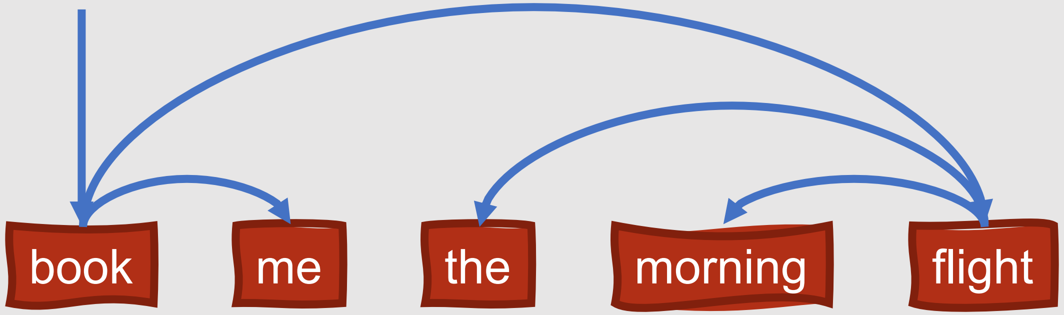
Input Buffer

Stack

root

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