LING/C SC 581:
Advanced Computational Linguistics

Lecture 30

Today's Topic

- More "dot moving"!
  - recall dotted rule notation (*for parsing state information*) from last time

Bottom-Up Parsing

- ***LR(0) parsing***
  - An example of **bottom-up** tabular parsing
  - 0 = zero symbols of lookahead, generally *N* (*a bit like the left corner idea*)

  - Similar to the **top-down Earley algorithm** described in the textbook in that it uses the idea of dotted rules

  - *finite state automata revisited*…

Tabular Parsing

- **e.g. LR(*k*)** (Knuth, 1960)
  - *invented for efficient parsing of programming languages*
  - **disadvantage**: a potentially huge number of states can be generated when the number of rules in the grammar is large
  - *can be applied to natural languages* (Tomita 1985)
  - build a Finite State Automaton (FSA) from the grammar rules, then add a stack
- **tables encode the grammar (FSA)**
  - grammar rules are compiled, we no longer interpret the grammar rules directly
- **Parser = Table + Push-down Stack**
  - table entries contain instruction(s) that tell what to do at a given state

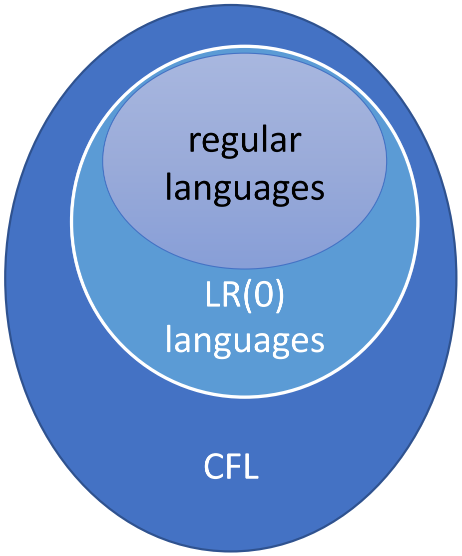
    ... *possibly factoring in lookahead*
  - stack data structure deals with maintaining the history of computation and recursion

Tabular Parsing

- **Shift-Reduce Parsing**
  - **example**
    - **LR(0)**
      - left to right
      - **bottom-up**
      - (0) no lookahead (input word)
    - **Three possible machine actions**
      - ***Shift***: read an input word
        - i.e. advance current input word pointer to the next word
      - ***Reduce***: complete a nonterminal
        - i.e. complete parsing a grammar rule
      - ***Accept***: complete the parse
        - i.e. start symbol (e.g. S) derives the terminal string

Tabular Parsing

- **LR(0) Parsing**
  - L(G) = LR(0)
    - *i.e. the language generated by grammar G is LR(0)*
    if there is a unique instruction per state
    (or no instruction = error state)
    LR(0) is a proper subset of context-free languages
  - **note**
    - human language tends to be ambiguous
    - there are likely to be multiple or conflicting actions per state
    - *if we are using Prolog, we can let Prolog's computation rule handle it*
      - *via Prolog backtracking*



Tabular Parsing

- **Dotted rule notation**
  - "dot" *used to track the progress of a parse through a phrase structure rule*
- **Examples:**
  - `vp --> vbd . np`

    means we've seen *v* and predict *np*
  - `np --> . dt nn`

    means we're predicting a *dt* (followed by *nn*)
  - `vp --> vp pp.`

    means we've completed a *vp* (with *pp* modification)

- **state**
  - a set of dotted rules encodes the state of the parse
  - *set of dotted rules = name of the state*
- **kernel**
  - `vp --> vbd . np`
  - `vp --> vbd .`
- **completion** (of predict NP)
  - `np --> . dt nn`
  - `np --> . nnp`
  - `np --> . np cp`

Tabular Parsing

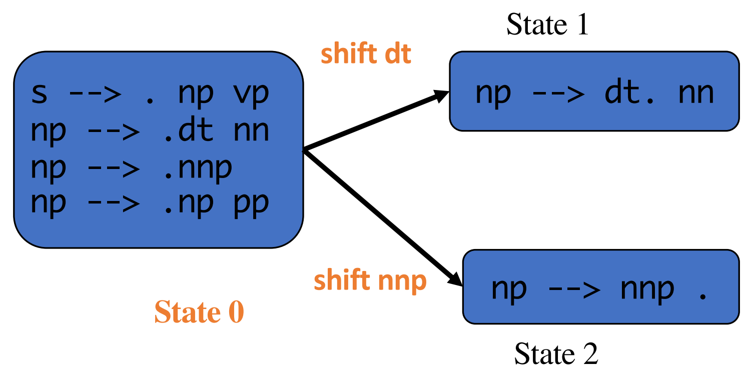
compute all possible states through advancing the dot

- **Example**:
- (Assume *dt is* next in the input)
  - `vp --> vbd . np`
  - `vp --> vbd .          (eliminated)`
  - `np --> dt . nn`
  - `np --> . nnp   (eliminated)`
  - `np --> . np cp`

Tabular Parsing

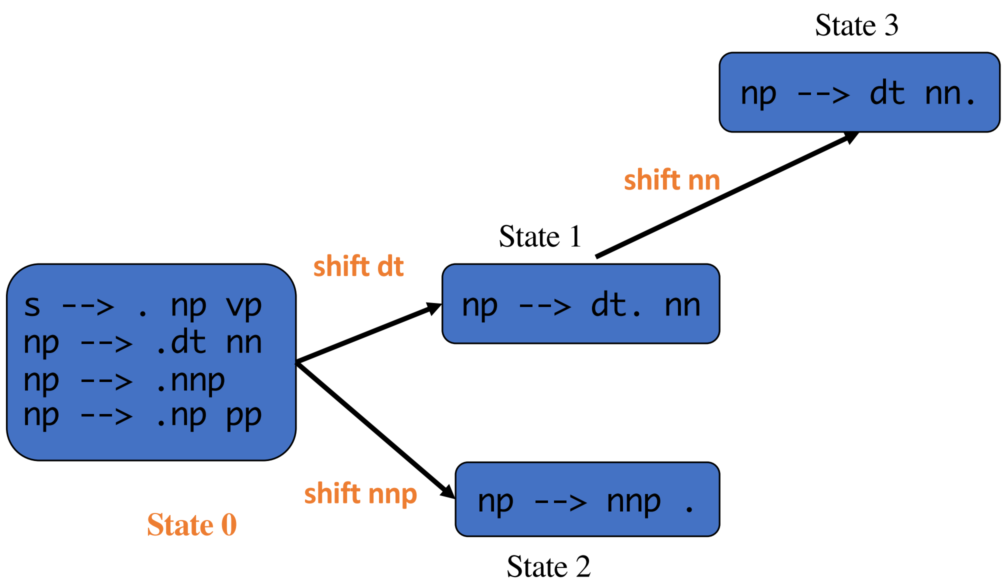
- **Dotted rules**
- **Example:**
  - **State 0**:
  - `s  --> .np vp`
  - `np --> .dt nn`
  - `np --> .nnp`
  - `np --> .np pp`
  - **possible actions**
    - shift *dt* and go to new state
    - shift *nnp* and go to new state

- Creating new states



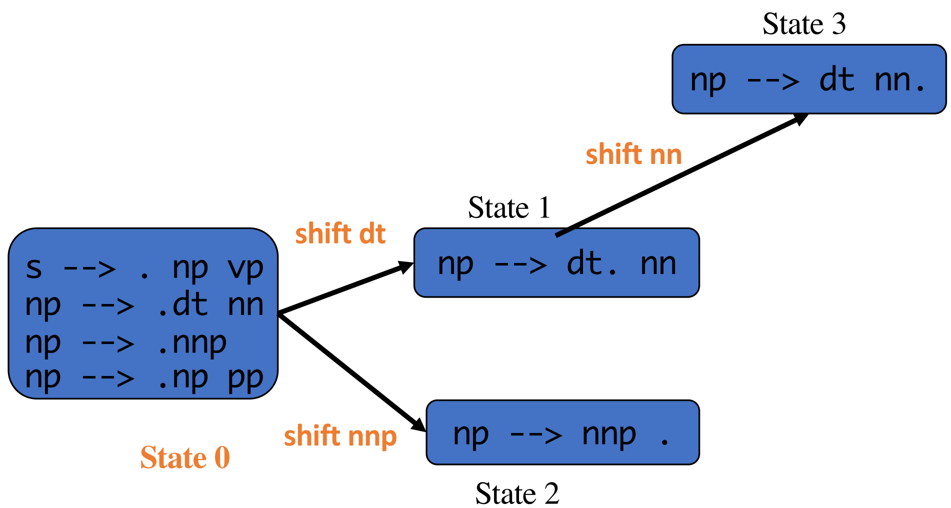
Tabular Parsing

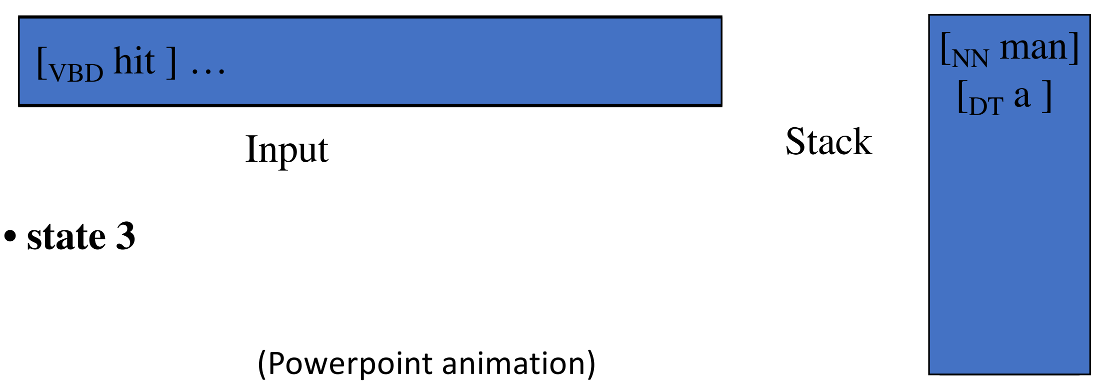
- **State 1: Shift *nn*, goto State 3**



Tabular Parsing

- **Shift**
  - take input word, and
  - put it on the stack



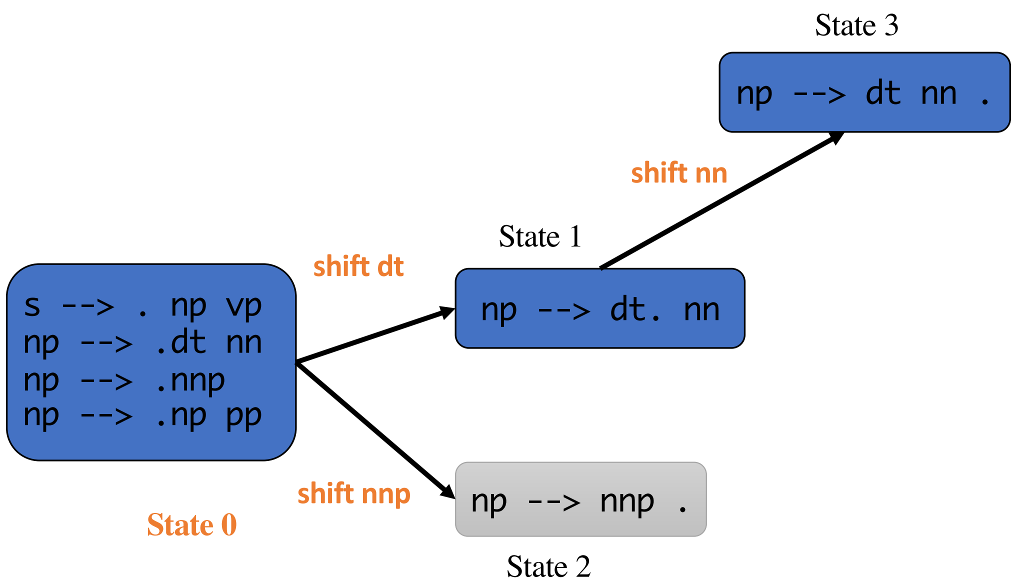


- **state 3**

(Powerpoint animation)

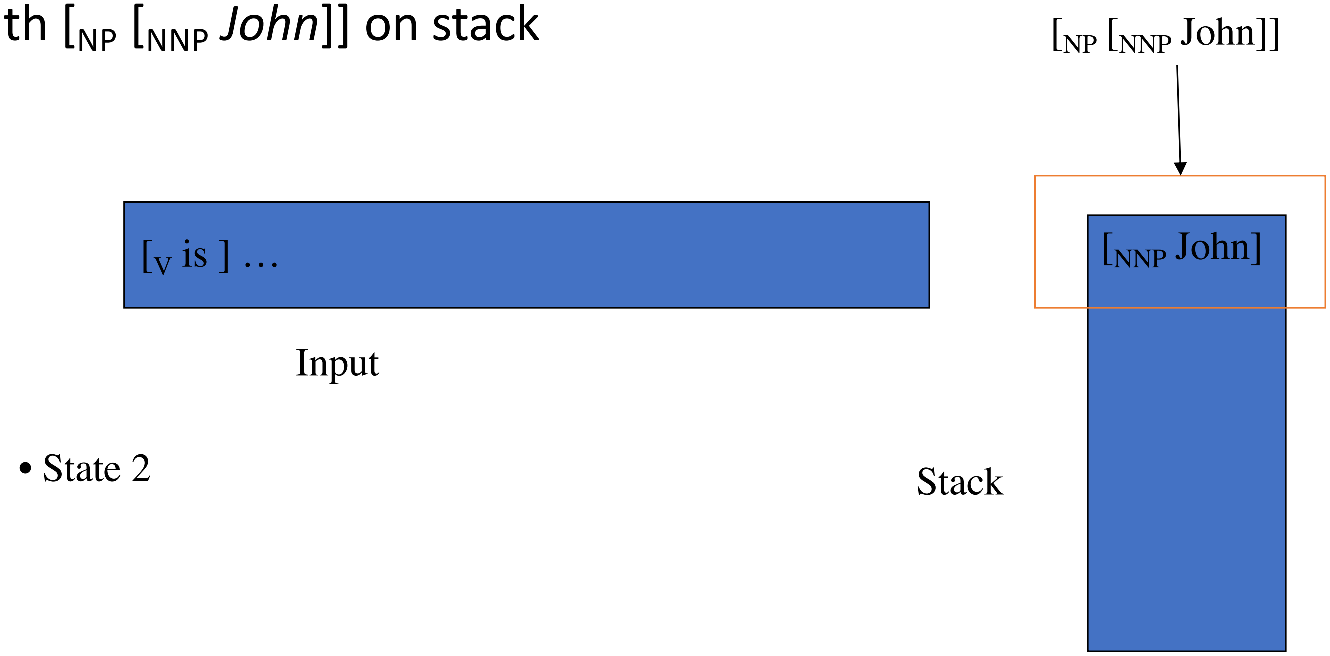
Tabular Parsing

- **State 2**: Reduce action np --> nnp .



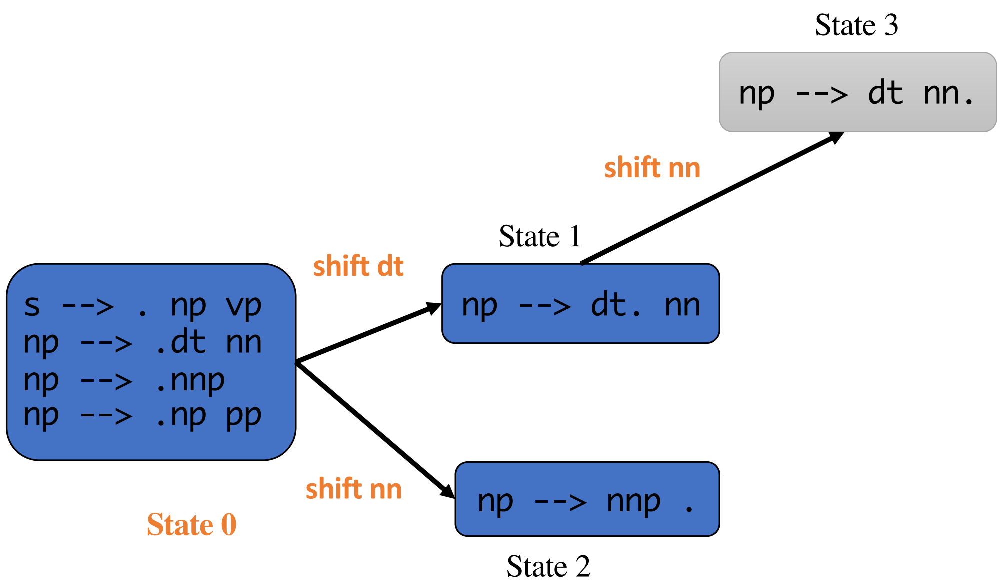
Tabular Parsing

- **Reduce** NP -> NNP .
  - pop [$_{NNP}$ *John*] off the stack, and
  - replace with [$_{NP}$ [$_{NNP}$ *John*]] on stack



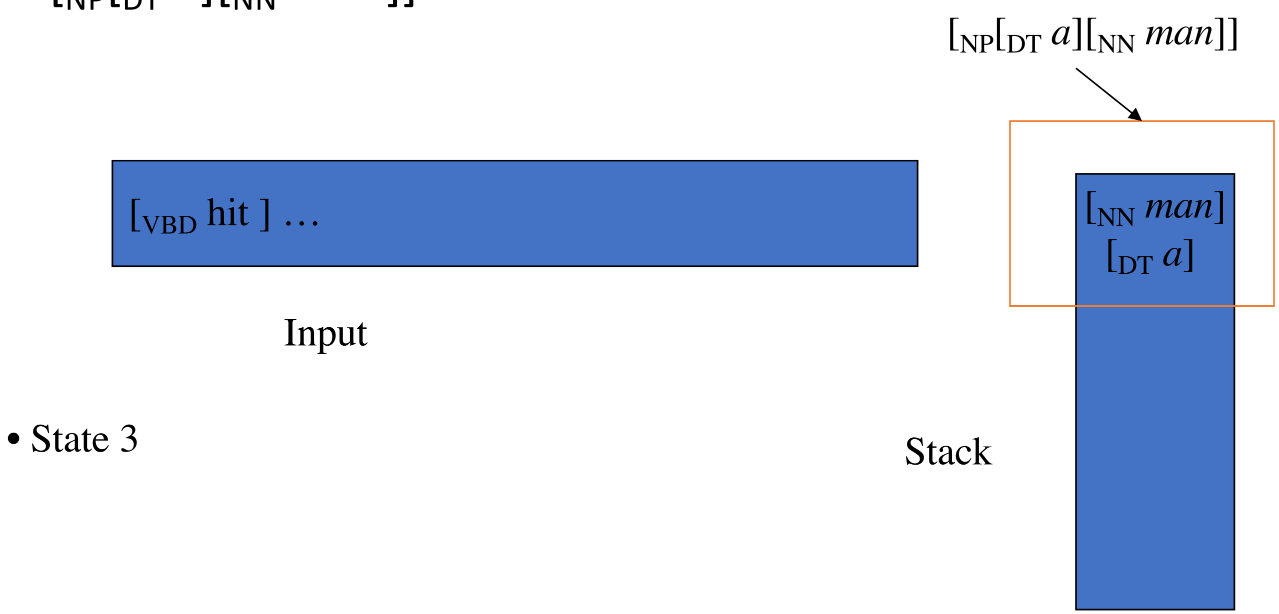
Tabular Parsing

- **State 3:** Reduce np --> dt nn.



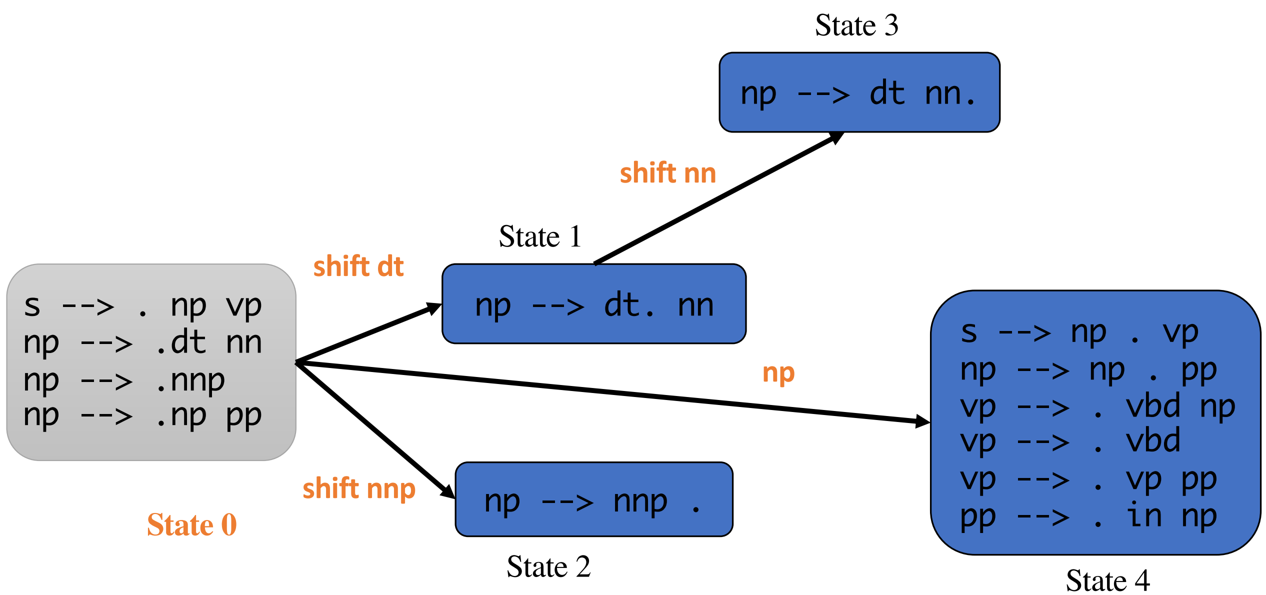
Tabular Parsing

- **Reduce** NP -> DT NN .
  - pop [$_{\text{NN}}$ *man*] and [$_{\text{DT}}$ *a*] off the stack
  - replace with [$_{\text{NP}}$[$_{\text{DT}}$ *a*][$_{\text{NN}}$ *man*]]



Tabular Parsing

- **State 0:** Transition NP



Tabular Parsing

- **for both states 2 and 3**
  - NP -> NNP .   (reduce NP -> NNP)
  - NP -> DT NN .   (reduce NP -> DT NN)
- **after Reduce NP operation**
  - **goto** state 4

- **notes**:
  - states are unique
  - grammar is finite
  - procedure generating states must terminate since the number of possible dotted rules is finite
  - no left recursion problem (*bottom-up means input driven*)

Tabular Parsing

- It's a table! (= **FSA**)

State	Action	Goto
0	Shift DT Shift NNP	1 2
1	Shift NN	3
2	Reduce NP --> NNP	4
3	Reduce NP --> DT NN	4
4	…	…

Tabular Parsing

- **Observations**
  1. ***table is sparse***
     - **Example:**
       - State 0, Input: [$_{VBD}$ ..]
       - parse fails immediately
  2. ***in a given state, input may be irrelevant***
     - **Example:**
       - State 2 (there is no shift operation)
  3. ***there may be action conflicts***
     - **Example:**
       - State 0: shift DT, shift NNP (*only if word is ambiguous*…)
     - **more interesting cases**
       - shift-reduce and reduce-reduce conflicts

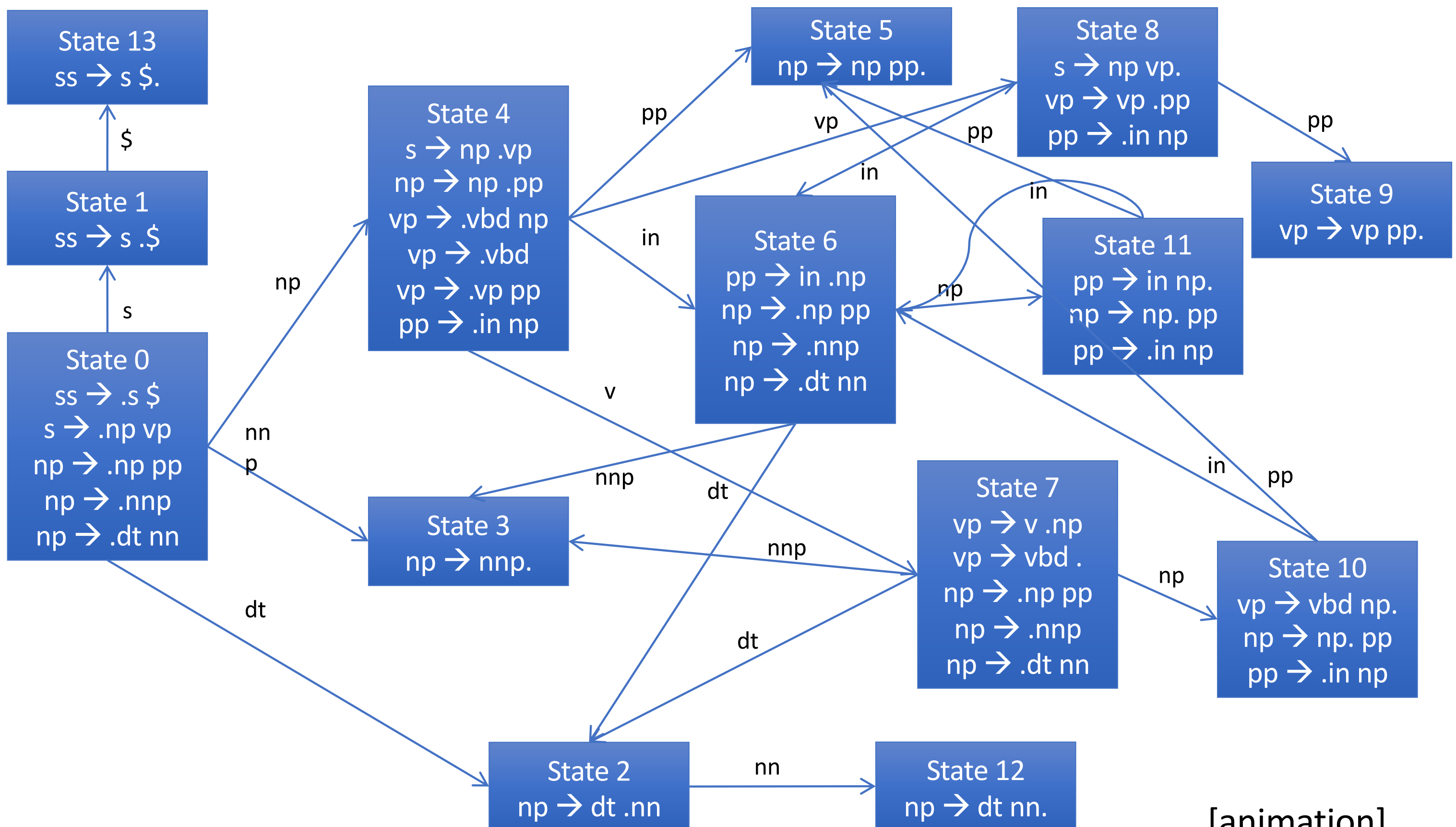
Tabular Parsing

- **finishing up**
  - an extra initial rule is usually added to the grammar
  - `SS --> S . $`
    - `SS` = start symbol
    - `$` = end of sentence marker
  - **input**:
    - *milk is good for you $*
  - **accept action**
    - discard $ from input
    - return element at the top of stack as the parse tree

LR Parsing in Prolog

- **Recap**
  - **finite state machine technology + a stack**
    - *each state represents a set of dotted rules*
    - **Example:**
      - `s --> . np vp`
      - `np --> .dt nn`
      - `np --> .nnp`
      - `np --> .np pp`
    - we transition, i.e. move, from state to state by advancing the “dot” over the possible terminal and nonterminal symbols

LR State Machine



[animation]

Build Actions

- **two main actions**
  - ***Shift***
    - move a word from the input onto the stack
    - Example:
      - *read a word with POS tag d*
      - `np --> .dt nn`
  - ***Reduce***
    - build a new constituent
    - Example:
      - *build a new NP*
      - `np --> dt nn.`

Lookahead

- **LR(1)**
  - a shift/reduce tabular parser
  - *using one (terminal) lookahead symbol*
  - *(like the left corner idea)*
- **decide on whether to take a reduce action depending on**
  - *state* x *next input symbol*
    - **Example**
      - *select the valid reduce operation consulting the next word*
      - cf. LR(0): *select an action based on just the current state*

Lookahead

- **potential advantage**
  - the input symbol may partition the action space
  - resulting in fewer conflicts
    - *provided the current input symbol can help to choose between possible actions*
- **potential disadvantages**
  1. larger finite state machine
     - more possible dotted rule/lookahead combinations than just dotted rule combinations
  2. might not help much
     - depends on the grammar
  3. more complex (off-line) computation
     - building the LR machine gets more complicated

Lookahead

- **formally**
  - `X --> α.Yβ, L`
    - `L` = lookahead set
    - `L` = set of possible terminals that can follow `X`
    - α,β (possibly empty) strings of terminal/non-terminals
- **Example:**
  - *State 0*
    - `ss-->.s $       [[]]`
    - `s-->.np vp      [$]`
    - `np-->.dt nn     [in, vbd]`
    - `np-->.nnp       [in, vbd]`
    - `np-->.np pp     [in, vbd]`

Lookahead

- **Central Idea**
    - *for propagating lookahead in state machine*
  - if dotted rule is complete,
  - **lookahead** informs parser about what the next terminal symbol should be

- **Example:**
  - `NP --> Dt NN. , L`
  - *reduce by NP rule only if current input symbol is in lookahead set* `L`

LR Parsing

- **In fact**
  - LR-parsers are generally acknowledged to be the fastest parsers
    - *especially when combined with the* ***chart technique*** *(table: dynamic programming)*
  - **reference**
    - (Tomita, 1985)
  - **textbook**
    - **Earley's algorithm**
    - uses chart
    - but follows the dotted-rule configurations **dynamically at parse-time**
    - instead of ahead of time (*so slower than LR*)