Quantifiers

- Not all noun phrases (NPs) are (by nature) directly referential like names
- **Quantifiers**: “*something to do with indicating the quantity of something*”
- **Examples**:
  - every child
  - nobody
  - two dogs
  - several animals
  - most people

`nobody has seen a unicorn`

means roughly (*Prolog-style*):

```prolog
?- setof(X,(person(X), seen(X,Y), unicorn(Y)),Set),cardinality(Set,0).
```
Quantifiers

- Database

nobody has seen a unicorn
  means roughly (Prolog-style):
  ?- setof(X,(person(X), seen(X,Y), unicorn(Y)),Set),cardinality(Set,0).

- setof vs. findall (recall last lecture)

Fix:
Quantifiers

• Semantic compositionality:
  – *elements of a sentence combine in piecewise fashion to form an overall (propositional) meaning for the sentence*

• Example:
  – (4) Every baby cried
    – **Word**      **Meaning**
    – cried        cried(X).
    – baby         baby(X).
    – **every**     ?
    – every baby cried      *proposition* (True/False)
    –                     *that can be evaluated for a given situation*
Quantifiers

- **Scenario (Possible World):**
  - Suppose there are three babies...
    - baby(noah).
    - baby(merrill).
    - baby(dani).
  - All three cried
    - cried(noah).
    - cried(merrill).
    - cried(dani).
  - Only Dani jumped
    - jumped(dani).
  - Noah and Dani swam
    - swam(noah).
    - swam(dani).

<table>
<thead>
<tr>
<th>(6)</th>
<th>every baby</th>
<th>exactly one baby</th>
<th>most babies</th>
</tr>
</thead>
<tbody>
<tr>
<td>cried</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>jumped</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>swam</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

- **think of quantifiers as “properties-of-properties”**
- every_baby(P) is a proposition
- P: property
- every_baby(P) **true** for P=cried
- every_baby(P) **false** for P=jumped and P=swam
Quantifiers

• think of quantifiers as “properties-of-properties”
  – every_baby(P) true for P=cried
  – every_baby(P) false for P=jumped and P=swam

• Generalized Quantifiers
  – the idea that quantified NPs represent sets of sets
  – this idea is not as weird as it sounds
  – we know
    • every_baby(P) is true for certain properties
  – view
    • every_baby(P) = set of all properties P for which this is true
  – in our scenario
    • every_baby(P) = {cried}
  – we know cried can also be view as a set itself
    • cried = set of individuals who cried
  – in our scenario
    • cried = {noah, merrill, dani}
Quantifiers

• how do we define the expression every_baby(P)?
• (Montague-style)
  every_baby(P) is shorthand for
  – for all individuals X, baby(X) -> P(X)
  – ->: if-then (implication : logic symbol)
• written another way
  (lambda calculus-style):
  – \( \lambda P. [\forall X. [\text{baby}(X) \rightarrow P(X)]] \)
  – \( \forall: \text{for all} \) (universal quantifier: logic symbol)

• Example:
  – every baby walks
    • for all individuals X, baby(X) -> walks(X)
    more formally
  – \([_{NP} \text{every baby}] [_{VP} \text{walks}]\)
    • \( \lambda P. [\forall X. [\text{baby}(X) \rightarrow P(X)]](\text{walks})\)
    • \( \forall X. [\text{baby}(X) \rightarrow \text{walks}(X)]\)
Quantifiers

• **how do we define this Prolog-style?**

  • Example:
    - every baby walks
    - \[NP\] every baby \[VP\] walks
      - \(\lambda P. [\forall X (\text{baby}(X) \rightarrow P(X))] (\text{walks})\)
      - \(\forall X (\text{baby}(X) \rightarrow \text{walks}(X))\)

  • **Possible World (Prolog database):**
    - \(-\text{ dynamic baby/1.} (\text{allows us to modify the baby database online})\)
    - baby(a). baby(b).
    - walks(a). walks(b). walks(c).
    - individual(a). individual(b). individual(c).

  • **What kind of query would you write?**

  • **One Possible Query (every means there are no exceptions):**
    - \(?- \\text{ \\+ (baby(X), \\+ walks(X))}. (\text{NOTE: may need a space between \\+ and ( here)}
      - Yes (TRUE)
    - \(?- \text{ baby(X), \\+ walks(X)}.\)
      - No
    - \(?- \text{ assert(baby(d))}.\)
    - \(?- \text{ baby(X), \\+ walks(X)}.\)
      - \(X = d ;\)
      - Yes

**Using no exception idea that \(\forall X P(X)\) is the same as \(-\exists X \neg P(X)\)**
**\(\exists = \text{“there exists” (quantifier)} (implicitly: all Prolog variables are existentially quantified variables)**
Recall: *Truth Tables*

- De Morgan’s Rule
- \( \neg(P \lor Q) = \neg P \land \neg Q \)

<table>
<thead>
<tr>
<th>(P)</th>
<th>(Q)</th>
<th>(P \lor Q)</th>
<th>(\neg(P \lor Q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\neg P)</th>
<th>(\land )</th>
<th>(\neg Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

\(\neg P \land \neg Q = T\) only when both \(P\) and \(Q\) are \(F\)

Hence, \(\neg(P \lor Q)\) is equivalent to \(\neg P \land \neg Q\)
Conversion into Prolog

Note:
\+ (baby(X), \+walks(X)) is Prolog for ∀X (baby(X) -> walks(X))

Steps:
- ∀X (baby(X) -> walks(X))
- ∀X (¬baby(X) v walks(X))
  • (since P->Q = ¬PvQ, see truth tables from two lectures ago)
- ¬∃X ¬(¬baby(X) v walks(X))
  • (since ∀X P(X) = ¬∃X ¬P(X), no exception idea)
- ¬∃X (baby(X) ∧ ¬walks(X))
  • (by De Morgan’s rule, see truth table from last slide)
- ¬(baby(X) ∧ ¬walks(X))
  • (can drop ∃X since all Prolog variables are basically existentially quantified variables)
- \+ (baby(X) ∧ \+walks(X))
  • (\+ = Prolog negation symbol)
- \+ (baby(X), \+walks(X))
  • (, = Prolog conjunction symbol)
Quantifiers

• how do we define this Prolog-style?

• Example:
  – every baby walks
  – \([_{\text{NP}} \text{every baby}] \ [_{\text{VP}} \text{walks}]\)
    • \(\lambda P. [\forall X. [\text{baby}(X) \rightarrow P(X)]](\text{walks})\)
    • \(\forall X. [\text{baby}(X) \rightarrow \text{walks}(X)]\)

• Another situation (Prolog database):
  – :- dynamic baby/1.
  – :- dynamic walks/1.

• Does \(?- \ + \ (\text{baby}(X), \ + \ \text{walks}(X)).\) still work?

• Yes because
  – \(?- \text{baby}(X), \ + \ \text{walks}(X).\)
  – No
cannot be satisfied
Quantifiers

- **how do we define the expression every_baby(P)?**
- (Montague-style)
  - every_baby(P) is shorthand for
    - $\lambda P. [\forall X. \text{baby}(X) \rightarrow P(X)]$

- (Barwise & Cooper-style)
  - think directly in terms of sets
  - *leads to another way of expressing the Prolog query*

- **Example**: every baby walks
  - $\{X: \text{baby}(X)\}$ *set of all X such that baby(X) is true*
  - $\{X: \text{walks}(X)\}$ *set of all X such that walks(X) is true*

- **Subset relation (⊆)**
  - $\{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\}$ *the “baby” set must be a subset of the “walks” set*
Quantifiers

(Barwise & Cooper-style)
• think directly in terms of sets
• leads to another way of expressing the Prolog query

• Example: every baby walks
• \{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\} \text{ the “baby” set must be a subset of the “walks” set}

• How to express this as a Prolog query?

• Queries:
• \texttt{?- setof(X,baby(X),L1).} \textit{L1 is the set of all babies in the database}
• \texttt{?- setof(X,walks(X),L2).} \textit{L2 is the set of all individuals who walk}

 Need a Prolog definition of the subset relation. This one, for example:
\begin{verbatim}
subset([],_).
subset([X|L1],L2) :- member(X,L2), subset(L1,L2).
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
\end{verbatim}
Quan%fiers

• **Example:** every baby walks
  
  \( \{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\} \)  
  
  the “baby” set must be a subset of the “walks” set

• **Assume the following definitions are part of the database:**
  
  \[
  \begin{align*}
  \text{subset}([\ ],\_). \\
  \text{subset}([X|\_ \_],L) :& - \text{member}(X,L). \\
  \text{member}(X,[X|\_ \_]) :& - \text{member}(X,L). \\
  \text{member}(X,[\_\_L]) :& - \text{member}(X,L).
  \end{align*}
  \]

• **Prolog Query:**

  • ?- setof(X,baby(X),L1), setof(X,walks(X),L2), subset(L1,L2).

  • **True for world:**

  - baby(a).  
  - baby(b).
  - walks(a).  
  - walks(b).  
  - walks(c).

  L1 = [a,b]
  L2 = [a,b,c]
  ?- subset(L1,L2) is true

  • **False for world:**

  - baby(a).  
  - baby(b).  
  - baby(d).
  - walks(a).  
  - walks(b).  
  - walks(c).

  L1 = [a,b,d]
  L2 = [a,b,c]
  ?- subset(L1,L2) is false
Quantifiers

• **Example:** *every baby walks*
  \[ \forall X \text{ (baby}(X) \rightarrow \text{walks}(X)) \]

• (Montague-style) \[ \{X: \text{baby}(X)\} \subseteq \{X: \text{walks}(X)\} \]

• (Barwise & Cooper-style) \[ \{X: \text{baby}(X)\} \subseteq \{X: \text{P}(X)\} \]

• **how do we define every_baby(P)?**
  
  • (Montague-style) \[ \lambda P. [\forall X \text{ (baby}(X) \rightarrow P(X))] \]
  
  • (Barwise & Cooper-style) \[ \{X: \text{baby}(X)\} \subseteq \{X: \text{P}(X)\} \]

• **how do we define every?**
  
  • (Montague-style) \[ \lambda P_1. [\lambda P_2. [\forall X \text{ (P}_1(X) \rightarrow P_2(X))]] \]
  
  • (Barwise & Cooper-style) \[ \{X: P_1(X)\} \subseteq \{X: P_2(X)\} \]
Quantifiers

• how do we define the expression *every*?
• (Montague-style) \( \lambda P_1. [\lambda P_2. [\forall X (P_1(X) \rightarrow P_2(X))] ] \)

• *Let’s look at computation in the lambda calculus...*
• **Example:** *every man likes John*
  – **Word** *Expression*
  – *every* \( \lambda P_1. [\lambda P_2. [\forall X (P_1(X) \rightarrow P_2(X))] ] \)
  – *man* man
  – *likes* \( \lambda Y. [\lambda X. [ X \text{ likes } Y]] \)
  – *John* John
• **Syntax:** \([S [NP [Q every][N man]][VP [V likes][NP John]]]]\)
Quantifiers

• **Example:** \(\text{[S [NP [Q every][N man]][VP [V likes][NP John]]]}\)
  
  - **Word**
    - *every* \(\lambda P_1.\lambda P_2.[\forall X (P_1(X) \rightarrow P_2(X))]\)
    - *man* \(\text{man}\)
    - *likes* \(\lambda Y.\lambda X.[X \text{ likes } Y]\)
    - *John* \(\text{John}\)

• **Logic steps:**
  
  \[
  \begin{align*}
  &\{Q \text{ every}\}[\text{N man}] \quad \lambda P_1.\lambda P_2.[\forall X (P_1(X) \rightarrow P_2(X))](\text{man})\\
  &\{Q \text{ every}\}[\text{N man}] \quad \lambda P_2.[\forall X (\text{man}(X) \rightarrow P_2(X))]\\
  &\{VP [V \text{ likes}]\}[\text{NP John}] \quad \lambda Y.\lambda X.[X \text{ likes } Y](\text{John})\\
  &\{VP [V \text{ likes}]\}[\text{NP John}] \quad \lambda X.[X \text{ likes } \text{John}]\\
  &\{S [NP [Q \text{ every}]\}[\text{N man}]\}[\text{VP [V \text{ likes}]\}[\text{NP John}]])\\
  &\quad \lambda P_2.[\forall X (\text{man}(X) \rightarrow P_2(X))](\lambda X.[X \text{ likes } \text{John}])\\
  &\quad \forall X (\text{man}(X) \rightarrow \lambda X.[X \text{ likes } \text{John}](X))\\
  &\quad \forall X (\text{man}(X) \rightarrow [X \text{ likes } \text{John}])
  \end{align*}
\]
Quantifiers

• Prolog is kinda first order logic ...
  – **no** quantifier variables
Quanifiers

• Example:
  – $\lambda P_1.[\lambda P_2.[\forall X (P_1(X) \rightarrow P_2(X))]]$
  lambda(P1,lambda(P2,\+ (P1(X), \+ P2(X))))

• Example:
  – $\{X: P(X)\}$
  setof(X,P(X),Set)
Quantifiers

- Example:
  - $\{X: P(X)\}$
  - Illegal: $\text{setof}(X, P(X), \text{Set})$
  - Alternate: $\text{setof}(X, \text{call}(P, X), \text{Set})$

Database

```
1 person(a).
2 person(b).
3 person(c).
?- call(person, X).
   X = a ;
?- P = person, setof(X, call(P, X), Set).
   P = person,
   Set = [a, b, c].
```

`call(:Goal)`  
Invoke `Goal` as a goal. Note that clauses may have variables as subclauses, which is identical to `call/1`.

`call(:Goal, +ExtraArg1, ...)`
Append `ExtraArg1, ExtraArg2, ...` to the argument list of `Goal` and call the result. For example, `call(plus(1), 2, X)` will call `plus(1, 2, X)`, binding `X` to 3.

The `call/[2..]` construct is handled by the compiler, which implies that redefinition as a predicate has no effect. The predicates `call/[2-6]` are defined as real predicates, so they can be handled by interpreted code.
Quantifiers

• Example:
  \[ \lambda P_1.[\lambda P_2.[\forall X. (P_1(X) \rightarrow P_2(X))] \]

  Illegal: \[ \text{lambda}(P_1, \text{lambda}(P_2, (\text{\texttt{\textbackslash +}} (P_1(X), \text{\texttt{\textbackslash +}} P_2(X)))))) \]

  Alternate: \[ \text{lambda}(P_1, \text{lambda}(P_2, (\text{\texttt{\textbackslash +}} (\text{call}(P_1,X), \text{\texttt{\textbackslash +}} \text{call}(P_2,X)))))) \]
Quantifiers

Part 3: Coordination

• Extend the grammars to handle
  – Every man and every woman likes John
Other Quantifiers

- Other quantifiers can also be expressed using set relations between two predicates:

  Example:
  
  \[ \text{no}: \{X: P_1(X)\} \cap \{Y: P_2(Y)\} = \emptyset \]

  \( \cap \) = set intersection

  \( \emptyset \) = empty set

  \textit{no man smokes}
  
  \[ \{X: \text{man}(X)\} \cap \{Y: \text{smokes}(Y)\} = \emptyset \]

  should evaluate to true for all possible worlds where there is no overlap between men and smokers
Other Quantifiers

- Other quantifiers can also be expressed using set relations between two predicates:

  Example:

  \[
  \text{some: } \{X: P_1(X)\} \cap \{Y: P_2(Y)\} \neq \emptyset \\
  \cap = \text{set intersection} \\
  \emptyset = \text{empty set}
  \]

  \text{some men smoke}

  \[
  \{X: \text{man}(X)\} \cap \{Y: \text{smokes}(Y)\} \neq \emptyset
  \]
Names as Generalized Quantifiers

- we’ve mentioned that names directly refer
  
  *here is another idea...*

- **Conjunction**
  - $X \text{ and } Y$
  - both $X$ and $Y$ have to be of the same type
  - *in particular,* semantically...
  - we want them to have the same semantic type

- **what is the semantic type of every baby?**

**Example**

- every baby and John likes ice cream
  - $\forall x (\text{baby}(x)) \land \forall y (\text{likes}(y, \text{ice cream}))$

- every baby likes ice cream
  - $\forall x (\text{baby}(x)) \subseteq \forall y (\text{likes}(y, \text{ice cream}))$

- John likes ice cream
  - $\exists y (\text{likes}(y, \text{ice cream}))$

- want everything to be a set (to be consistent)
  - i.e. want to state something like
  - $\forall x (\text{baby}(x)) \cup \forall y (\text{john}(y)) \subseteq \forall y (\text{likes}(y, \text{ice cream}))$

- note: set union ($\cup$) is the translation of “and”
Downwards and Upwards Entailment (DE & UE)

- **Quantifier every** has semantics
  - \( \{X: \text{P}_1(X)\} \subseteq \{Y: \text{P}_2(Y)\} \)
  - e.g. every woman likes ice cream
  - \( \{X: \text{woman}(X)\} \subseteq \{Y: \text{likes}(Y,\text{ice\_cream})\} \)

- **Every** is DE for \( \text{P}_1 \) and UE for \( \text{P}_2 \)
- Examples:
  - (25) a. Every dog barks
  - b. Every Keeshond barks  (valid)
  - c. Every animal barks  (invalid)
    - semantically, “Keeshond” is a sub-property or subset with respect to the set “dog”
Downwards and Upwards Entailment (DE & UE)

• Quantifier **every** has semantics
  – \( \{X: P_1(X)\} \subseteq \{Y: P_2(Y)\} \)
  – e.g. every woman likes ice cream
  – \( \{X: \text{woman}(X)\} \subseteq \{Y: \text{likes}(Y, \text{ice\_cream})\} \)
• **Every** is DE for \( P_1 \) and UE for \( P_2 \)
• Examples:
  • (25) a. Every dog barks
  • d. **Every dog barks loudly** (invalid)
  • c. Every dog makes noise (valid)
    – semantically, “barks loudly” is a subset with respect to the set “barks”, which (in turn) is a subset of the set “makes noise”
Parse Methods

- **Basic Prolog grammar system:**
  - we've used Prolog's default computation rule to parse context-free grammars
  - This strategy is known as a top-down, depth-first search strategy
  - *There are many other methods ...*

**problems**
- left-recursion
  - gives termination problems
- no bottom-up filtering
  - *inefficient*
- left-corner idea
Top-Down Parsing with Left-Corner Filtering

- **no bottom-up filtering**
  - left-corner idea
  - eliminate unnecessary top-down search
  - reduce the number of choice points (*amount of branching*)

- **example**
  - does this flight include a meal?

- **computation:**
  1. $s \rightarrow np, \ vp.$
  2. $s \rightarrow aux, \ np, \ vp.$
  3. $s \rightarrow vp.$
  - left-corner idea rules out 1 and 3
Left Corner Parsing

- need bottom-up filtering
  - filter top-down rule expansion using bottom-up information
  - current input is the bottom-up information
  - left-corner idea
- example
  - s(s(NP,VP)) --> np(NP), vp(VP).
  - what terminals can be used to begin this phrase?
    - answer: whatever can begin NP
      - np(np(D,N)) --> det(D), nominal(N).
      - np(np(PN)) --> propernoun(PN).
    - answer: whatever can begin Det or ProperNoun
      - det(det(that)) --> [that].
      - det(det(this)) --> [this].
      - det(det(a)) --> [a].
      - propernoun(propn(houston)) --> [houston].
      - propernoun(propn(twa)) --> [twa].
    - answer: 
      - {that,this,a,houston,twa}  "Left Corner"
Left Corner Parsing

- example
  - does this flight include a meal?

- computation
  1. \( s(s(NP,VP)) \rightarrow np(NP), vp(VP) \). LC: \{that, this, a, houston, twa\}
  2. \( s(s(Aux,NP,VP)) \rightarrow aux(Aux), np(NP), vp(VP) \). LC: \{does\}
  3. \( s(s(VP)) \rightarrow vp(VP) \). LC: \{book, include, prefer\}
  - only rule 2 is compatible with the input
  - match first input terminal against left-corner (LC) set for each possible matching rule
  - left-corner idea prunes away or rules out options 1 and 3
Left Corner Parsing

- **DCG Rules**
  1. $s(s(NP,VP)) \rightarrow np(NP), vp(VP)$. LC: \{that, this, a, houston, twa\}
  2. $s(s(Aux,NP,VP)) \rightarrow aux(Aux), np(NP), vp(VP)$. LC: \{does\}
  3. $s(s(VP)) \rightarrow vp(VP)$. LC: \{book, include, prefer\}

- **left-corner database facts**
  - `$lc$ (rule#, [word|\_], [word|\_]).$
  - $lc(1, [that|L], [that|L])$. $lc(2, [does|L], [does|L])$. $lc(3, [book|L], [book|L])$. $lc(1, [a|L], [a|L])$. $lc(3, [include|L], [include|L])$. $lc(1, [houston|L], [houston|L])$. $lc(3, [prefer|L], [prefer|L])$. $lc(1, [twa|L], [twa|L])$. 

- **rewrite Prolog rules to check input against $lc$**
  1. $s(s(NP,VP)) \rightarrow lc(1), np(NP), vp(VP)$.
  2. $s(s(Aux,NP,VP)) \rightarrow lc(2), aux(Aux), np(NP), vp(VP)$.
  3. $s(s(VP)) \rightarrow lc(3), vp(VP)$.
Lev	
  
  Corner	
  
  Parsing

• left-corner database facts
  - \% lc(rule#,[word|_],[word|_]).
  - lc(1,[that|L],[that|L]).
  - lc(2,[does|L],[does|L]).
  - lc(1,this|L],[this|L]).
  - lc(3,[book|L],[book|L]).
  - lc(1,[a|L],[a|L]).
  - lc(3,[include|L],[include|L]).
  - lc(1,[houston|L],[houston|L]).
  - lc(3,[prefer|L],[prefer|L]).
  - lc(1,[twa|L],[twa|L]).

• rewrite DCG rules to check input against lc/3
  1. s(s(NP,VP)) --> lc(1), np(NP), vp(VP).
  2. s(s(Aux,NP,VP)) --> lc(2), aux(Aux), np(NP), vp(VP).
  3. s(s(VP)) --> lc(3), vp(VP).

• DCG rules are translated into underlying Prolog rules:
  1. s(s(A,B), C, D) :- lc(1, C, E), np(A, E, F), vp(B, F, D).
  2. s(s(A,B,C), D, E) :- lc(2, D, F), aux(A, F, G), np(B, G, H), vp(C, H, E).
  3. s(s(A), B, C) :- lc(3, B, D), vp(A, D, C).
Left Corner Parsing

- **Summary:**
  - Given a context-free DCG
  - Generate left-corner database facts
    - `lc(rule#, [word|_], [word|_])`.
  - Rewrite DCG rules to check input against `lc`
    - `s(s(NP,VP)) --> lc(1), np(NP), vp(VP)`.
  - DCG rules are translated into underlying Prolog rules:
    - `s(s(A,B), C, D) :- lc(1, C, E), np(A, E, F), vp(B, F, D)`.

- **Note:**
  - This process can be done automatically (by program)
  - **not all rules need be rewritten**
  - **lexicon rules are direct left-corner rules**
  - **no filtering is necessary**
    - `det(det(a)) --> [a].`
    - `noun(noun(book)) --> [book].`
  - **i.e. no need to call lc as in**
    - `det(det(a)) --> lc(11), [a].`
    - `noun(noun(book)) --> lc(12), [book].`
Bottom-Up Parsing

• *LR(0)* parsing
  – An example of **bottom-up** tabular parsing

– Similar to the **top-down** Earley algorithm described in the textbook in that it uses the idea of dotted rules
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
  - 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)
- **LR 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**
  – \( L(G) = \text{LR}(0) \)
    • *i.e. the language generated by grammar} \( G \) * is \( \text{LR}(0) \)
      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
    • *can let Prolog’s computation rule handle it*
      – *i.e. use Prolog backtracking*
Tabular Parsing

• **Dotted Rule Notation**
  - “dot” *used to indicate the progress of a parse through a phrase structure rule*
  - **examples**
    • $\text{vp} \rightarrow \text{v} \cdot \text{np}$
      means we’ve seen v and predict np
    • $\text{np} \rightarrow \cdot \text{d np}$
      means we’re predicting a d (followed by np)
    • $\text{vp} \rightarrow \text{vp pp}$. mean we’ve completed a vp

• **state**
  - a set of dotted rules encodes the state of the parse

• **kernel**
  - $\text{vp} \rightarrow \text{v} \cdot \text{np}$
  - $\text{vp} \rightarrow \text{v} \cdot$

• **completion** (of predict NP)
  - $\text{np} \rightarrow \cdot \text{d n}$
  - $\text{np} \rightarrow \cdot \text{n}$
  - $\text{np} \rightarrow \cdot \text{np cp}$
Tabular Parsing

- compute possible states by advancing the dot
  - example:
    - (Assume $d$ is next in the input)
      - $vp \rightarrow v . np$
      - $vp \rightarrow v.$ *(eliminated)*
      - $np \rightarrow d . n$
      - $np \rightarrow . n$ *(eliminated)*
      - $np \rightarrow . np cp$
Tabular Parsing

- **Dotted rules**
  - example
    - State 0:
      - s $\rightarrow$ . np vp
      - np $\rightarrow$ .d np
      - np $\rightarrow$ .n
      - np $\rightarrow$.np pp
  - possible actions
    - **shift** d and go to new state
    - **shift** n and go to new state

- **Creating new states**
Tabular Parsing

- **State 1**: Shift N, goto State 2

  - S -> . NP VP
  - NP -> . D N
  - NP -> . N
  - NP -> . NP PP

  - NP -> D . N
  - NP -> N .

- State 0

- State 1

- State 2

- State 3
Tabular Parsing

- **Shift**
  - take input word, and
  - place on stack

- **state 3**
Tabular Parsing

- **State 2**: Reduce action NP -> N.

```
S -> . NP VP
NP -> . D N
NP -> . N
NP -> . NP PP

NP -> D N .

State 0

State 1

NP -> D . N

State 3

NP -> N .

State 2
```
Tabular Parsing

• **Reduce** NP -> N .
  – pop \([_N{\text{milk}}]\) off the stack, and
  – replace with \([_{NP}[_N{\text{milk}}]]\) on stack

```
[_{V \text{ is } ...}]
\text{Input}
```

• State 2

```
[_{NP \text{ milk}}]
\text{Stack}
```

[_{N \text{ milk}}]
Tabular Parsing

- **State 3**: Reduce NP -> D N .

S -> . NP VP
NP -> . D N
NP -> . N
NP -> . NP PP
Tabular Parsing

- **Reduce** $NP \rightarrow D N$.
  - pop $[N \text{ man}]$ and $[D \text{ a}]$ off the stack
  - replace with $[NP[D \text{ a}][N \text{ man}]]$

```
[\_N \text{ man}] \_{D \text{ a}]}[\_N \text{ man}]
```

```
Input

[\_V \text{ hit} ] ...
```

```
[\_N \text{ man}]
```

```
Stack

[\_D \text{ a} ]
```
Tabular Parsing

- **State 0**: Transition NP

  - **State 0**: $S \rightarrow \cdot \ NP \ VP$
    - $NP \rightarrow \cdot \ D \ N$
    - $NP \rightarrow \cdot \ N$
    - $NP \rightarrow \cdot \ NP \ PP$

  - **State 2**: $NP \rightarrow \cdot \ N$

  - **State 4**: $S \rightarrow \ NP \ . \ VP$
    - $NP \rightarrow \ NP \ . \ PP$
    - $VP \rightarrow \cdot \ V \ NP$
    - $VP \rightarrow \cdot \ V$
    - $VP \rightarrow \cdot \ VP \ PP$
    - $PP \rightarrow \cdot \ P \ NP$
Tabular Parsing

• for both states 2 and 3
  – NP -> N. (reduce NP -> N)
  – NP -> D N. (reduce NP -> D N)

• 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
## Tabular Parsing

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shift D</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Shift N</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Shift N</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Reduce NP -&gt; N</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Reduce NP -&gt; D N</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tabular Parsing

- **Observations**
  - *table is sparse*
    - example
      - State 0, Input: $[V ..]$  
      - parse fails immediately
  - *in a given state, input may be irrelevant*
    - example
      - State 2 (there is no shift operation)
  - *there may be action conflicts*
    - example
      - State 1: shift D, shift N

- **more interesting cases**
  - shift-reduce and reduce-reduce conflicts
Tabular Parsing

• **finishing up**
  – an extra initial rule is usually added to the grammar
  – \( SS \rightarrow S \ . \ $ \)
    • \( SS \) = 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
    • each state represents a set of dotted rules
      – example
        » S  --> . NP VP
        » NP  --> . D N
        » NP  --> . N
        » NP  --> . NP PP
  • we transition, i.e. move, from state to state by advancing the “dot” over terminal and nonterminal symbols
Build Actions

• two main actions
  – *Shift*
    • move a word from the input onto the stack
    • Example:
      – NP --> D N
  
  – *Reduce*
    • build a new constituent
    • Example:
      – NP --> D N.
• **Example:**
  - `?- parse([john,saw,the,man,with,a,telescope],X).`
  - `X =
    s(np(n(john)),vp(v(saw),np(np(d(the),n(man)),pp(p(with),np(d(a),n(telescope))))))) ;`
  - `X =
    s(np(n(john)),vp(vp(v(saw),np(d(the),n(man))),pp(p(with),np(d(a),n(telescope)))))) ;`
  - `no`
## LR(0) Goto Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
**LR(0) Action Table**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>S</td>
<td>D</td>
<td>A</td>
<td>S</td>
<td>N</td>
<td>R</td>
<td>NP</td>
<td>S</td>
<td>V</td>
<td>R</td>
<td>NP</td>
<td>S</td>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>A₂</td>
<td>S</td>
<td>N</td>
<td>S</td>
<td>P</td>
<td>S</td>
<td>N</td>
<td>S</td>
<td>N</td>
<td>R</td>
<td>S</td>
<td>R</td>
<td>VP</td>
<td>R</td>
<td>VP</td>
</tr>
<tr>
<td>A₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RV</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S = shift, R = reduce, A = accept

Empty cells = error states

Multiple actions = machine conflict

Prolog’s computation rule: *backtrack*
LR(0) Conflict Statistics

- **Toy grammar**
  - 14 states
  - 6 states
    - with 2 competing actions
    - states 11, 10, 8:
      - *shift-reduce conflict*
  - 1 state
    - with 3 competing actions
    - State 7:
      - *shift(d) shift(n) reduce(vp->v)*

![Bar chart showing number of states with conflicts](image)
LR Parsing

• in fact
  – LR-parsers are generally acknowledged to be the fastest parsers
    • using lookahead (current terminal symbol)
    • and when combined with the chart technique (memorizing subphrases in a table - dynamic programming)
  – textbook
    • Earley’s algorithm (13.4.2)
    • uses chart
    • but builds dotted-rule configurations dynamically at parse-time
    • instead of ahead of time (so slower than LR)