Why are parsing algorithms important?

- A linguistic theory is implemented in a formal system to generate the set of grammatical strings and rule out ungrammatical strings.
- Such a formal system has computational properties.
- One such property is a simple decision problem: given a string, can it be generated by the formal system (recognition).
- If it is generated, what were the steps taken to recognize the string (parsing).
Why are parsing algorithms important?

- Consider the recognition problem: find algorithms for this problem for a particular formal system.
- The algorithm must be decidable.
- Preferably the algorithm should be polynomial: enables computational implementations of linguistic theories.
- Elegant, polynomial-time algorithms exist for formalisms like CFG.

Top-down, depth-first, left to right parsing

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow Det \ N \\
NP & \rightarrow Det \ N \ PP \\
VP & \rightarrow V \\
VP & \rightarrow V \ NP \\
VP & \rightarrow V \ NP \ PP \\
PP & \rightarrow P \ NP \\
NP & \rightarrow I \\
Det & \rightarrow a \mid the \\
V & \rightarrow saw \\
N & \rightarrow park \mid dog \mid man \mid telescope \\
P & \rightarrow in \mid with
\end{align*}
\]
Top-down, depth-first, left to right parsing

- Consider the input string: *the dog saw a man in the park*
- \( S \ldots (S \ (NP \ VP)) \ldots (S \ (NP \ Det \ N) \ VP) \ldots (S \ (NP \ (Det \ the) \ N) \ VP) \ldots (S \ (NP \ (Det \ the) \ (N \ dog)) \ VP) \ldots \)
- \((S \ (NP \ (Det \ the) \ (N \ dog)) \ VP) \ldots (S \ (NP \ (Det \ the) \ (N \ dog)) \ (VP \ V \ NP \ PP)) \ldots (S \ (NP \ (Det \ the) \ (N \ dog)) \ (VP \ (V \ saw) \ NP \ PP)) \ldots \)
- \((S \ (NP \ (Det \ the) \ (N \ dog)) \ (VP \ (V \ saw) \ (NP \ Det \ N) \ PP)) \ldots \)
- \((S \ (NP \ (Det \ the) \ (N \ dog)) \ (VP \ (V \ saw) \ (NP \ (Det \ a) \ (N \ man)) \ (PP \ (P \ in) \ (NP \ (Det \ the) \ (N \ park)))))\)

Number of derivations

CFG rules \{ S \rightarrow S \ S , S \rightarrow a \}

<table>
<thead>
<tr>
<th>( n : a^n )</th>
<th>number of parses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
</tr>
<tr>
<td>8</td>
<td>429</td>
</tr>
<tr>
<td>9</td>
<td>1430</td>
</tr>
<tr>
<td>10</td>
<td>4862</td>
</tr>
<tr>
<td>11</td>
<td>16796</td>
</tr>
</tbody>
</table>
Number of derivations grows exponentially

\[ L(G) = a + \text{using CFG rules } \{ S \rightarrow SS, S \rightarrow a \} \]

Syntactic Ambiguity: (Church and Patil 1982)

- Algebraic character of parse derivations
- Power Series for grammar for coordination type of grammars (more general than PPs):
  \[ N \rightarrow \text{natural} \mid \text{language} \mid \text{processing} \mid \text{course} \]
  \[ N \rightarrow N N \]
- We write an equation for algebraic expansion starting from \( N \)
- The equation represents generation of each string in the language as the terms, and the number of different ways of generating the string as the coefficients:
  \[ N = \text{nat.} + \text{lang.} + \text{proc.} + \text{course} + \]
  \[ + \text{nat. lang.} + \text{nat. proc.} + \ldots \]
  \[ + 2(\text{nat. lang. proc.}) + 2(\text{lang. proc. course}) + \ldots \]
  \[ + 5(\text{nat. lang. proc. course}) + \ldots \]
  \[ + 14 \ldots \]
CFG Ambiguity

- Coefficients in previous equation equal the number of parses for each string derived from $E$
- These ambiguity coefficients are Catalan numbers:

$$Cat(n) = \frac{1}{n+1} \binom{2n}{n}$$

- $\binom{a}{b}$ is the binomial coefficient

$$\binom{a}{b} = \frac{a!}{b!(a-b)!}$$

Catalan numbers

- Why Catalan numbers? $Cat(n)$ is the number of ways to parenthesize an expression of length $n$ with two conditions:
  1. there must be equal numbers of open and close parens
  2. they must be properly nested so that an open precedes a close
Catalan numbers

- For an expression of with $n$ ways to form constituents there are a total of $2n$ choose $n$ parenthesis pairs, e.g. for $n = 2$, $\binom{4}{2} = 6$:
  
  $$\text{a}(\text{bc}), \text{ a)}\text{bc}(, \text{ )a}(\text{bc}, \text{ (ab)}\text{c, )ab(c, ab)}\text{c}(\text{)}$$

- But for each valid parenthesis pair, additional $n$ pairs are created that have the right parenthesis to the left of its matching left parenthesis, from e.g. above: $\text{a)}\text{bc}(, \text{ )a}(\text{bc}, \text{ )ab(c, ab)}\text{c}(\text{)}$

- So we divide $2n$ choose $n$ by $n + 1$:

  $$Cat(n) = \frac{\binom{2n}{n}}{n + 1}$$

<table>
<thead>
<tr>
<th>$n$</th>
<th>catalan($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>132</td>
</tr>
<tr>
<td>7</td>
<td>429</td>
</tr>
<tr>
<td>8</td>
<td>1430</td>
</tr>
<tr>
<td>9</td>
<td>4862</td>
</tr>
<tr>
<td>10</td>
<td>16796</td>
</tr>
</tbody>
</table>
Syntactic Ambiguity

- *Cat*(n) also provides exactly the number of parses for the sentence: *John saw the man on the hill with the telescope* (generated by the grammar given below, a different grammar will have different number of parses)

```
S → NP VP
NP → John | Det N
N → man | hill | telescope
VP → V NP
Det → the
VP → VP PP
NP → NP PP
NP → NP PP
P → P NP
V → saw
P → on | with
```

number of parse trees = Cat(2 + 1) = 5.
With 8 PPs: Cat(9) = 4862 parse trees
Syntactic Ambiguity

- For grammar on previous page, number of parse trees = \( \text{Cat}(2 + 1) = 5 \).
- Why \( \text{Cat}(2 + 1) \)?
  - For 2 PPs, there are 4 things involved: VP, NP, PP-1, PP-2
  - We want the items over which the grammar imposes all possible parentheses
  - The grammar is structured in such a way that each combination with a VP or an NP reduces the set of items over which we obtain all possible parentheses to 3
  - This can be viewed schematically as VP * NP * PP-1 * PP-2
    1. \((\text{VP} (\text{NP} (\text{PP-1 PP-2}))\))
    2. \((\text{VP} ((\text{NP PP-1}) \text{ PP-2}))\))
    3. \(((\text{VP NP}) (\text{PP-1 PP-2}))\))
    4. \(((\text{VP (NP PP-1)}) \text{ PP-2})\))
    5. \(((\text{VP NP}) \text{ PP-1) PP-2})\))
  - Note that combining PP-1 and PP-2 is valid because PP-1 has an NP inside it.

Syntactic Ambiguity

- Other sub-grammars are simpler. For chains of adjectives: *cross-eyed pot-bellied ugly hairy professor* We can write the following grammar, and compute the power series:

\[
\begin{align*}
\text{ADJP} & \rightarrow \text{adj ADJP} \mid \epsilon \\
\text{ADJP} & = 1 + \text{adj} + \text{adj}^2 + \text{adj}^3 + \ldots
\end{align*}
\]
Syntactic Ambiguity

Now consider power series of combinations of sub-grammars:

\[ S = \text{NP} \cdot \text{VP} \]

( The number of products over sales ... )

( is near the number of sales ... )

Both the NP subgrammar and the VP subgrammar power series have Catalan coefficients

Syntactic Ambiguity

The power series for the \( S \rightarrow \text{NP} \text{ VP} \) grammar is the multiplication:

\[
( N \sum_{i} \text{Cat}_{i} ( P \text{ N } )^{i} ) \cdot ( \text{is} \sum_{j} \text{Cat}_{j}( P \text{ N } )^{j} )
\]

In a parser for this grammar, this leads to a cross-product:

\[
L \times R = \{( l, r ) \mid l \in L \& r \in R \}
\]
Syntactic Ambiguity

- A simple change:
  
  \[
  \text{Is ( The number of products over sales ... )}
  \]
  
  \[
  ( \text{ near the number of sales ... })
  \]
  
  \[
  = \text{Is } N \sum_i \text{Cat}_i ( P N )^i \cdot \left( \sum_j \text{Cat}_j ( P N )^j \right)
  \]
  
  \[
  = \text{Is } N \sum_i \sum_j \text{Cat}_i \text{Cat}_j ( P N )^{i+j}
  \]
  
  \[
  = \text{Is } N \sum_{i+j} \text{Cat}_{i+j+1} ( P N )^{i+j}
  \]

Dealing with Ambiguity

- A CFG for natural language can end up providing exponentially many analyses, approx \( n! \), for an input sentence of length \( n \)

- Much worse than the worst case in the part of speech tagging case, which was \( n^m \) for \( m \) distinct part of speech tags

- If we actually have to process all the analyses, then our parser might as well be exponential

- Typically, we can directly use the compact description (in the case of CKY, the chart or 2D array, also called a forest)
Dealing with Ambiguity

- Solutions to this problem:
  - CKY algorithm: computes all parses in $O(n^3)$ time. Problem is that worst-case and average-case time is the same.
  - Earley algorithm: computes all parses in $O(n^3)$ time for arbitrary CFGs, $O(n^2)$ for unambiguous CFGs, and $O(n)$ for so-called bounded-state CFGs (e.g. $S \rightarrow aSa \mid bSb \mid aa \mid bb$ which generates palindromes over the alphabet $a, b$). Also, average case performance of Earley is better than CKY.
  - Deterministic parsing: only report one parse. Two options: top-down (LL parsing) or bottom-up (LR or shift-reduce) parsing.

Shift-Reduce Parsing

- Every CFG has an equivalent pushdown automata: a finite state machine which has additional memory in the form of a stack.
- Consider the grammar: $NP \rightarrow Det \ N$, $Det \rightarrow the$, $N \rightarrow dogs$
- Consider the input: the dogs
- Shift the first word the into the stack, check if the top $n$ symbols in the stack matches the right hand side of a rule in which case you can reduce that rule, or optionally you can shift another word into the stack.
Shift-Reduce Parsing

- reduce using the rule $Det \rightarrow the$, and push $Det$ onto the stack
- shift $dogs$, and then reduce using $N \rightarrow dogs$ and push $N$ onto the stack
- the stack now contains $Det, N$ which matches the rhs of the rule $NP \rightarrow Det \ N$ which means we can reduce using this rule, pushing $NP$ onto the stack
- If $NP$ is the start symbol and since there is no more input left to shift, we can accept the string
- Can this grammar get stuck (that is, there is no shift or reduce possible at some stage while parsing) on a valid string?
- What happens if we add the rule $NP \rightarrow dogs$ to the grammar?

Shift-Reduce Parsing

- Sometimes humans can be “led down the garden-path” when processing a sentence (from left to right)
- Such garden-path sentences lead to a situation where one is forced to backtrack because of a commitment to only one out of many possible derivations
- Consider the sentence:
  *The emergency crews hate most is domestic violence.*
- Consider the sentence:
  *The horse raced past the barn fell*
Shift-Reduce Parsing

- Once you process the word *fell* you are forced to reanalyze the previous word *raced* as being a verb inside a *relative clause*: *raced past the barn*, meaning *the horse that was raced past the barn*.
- Notice however that other examples with the same structure but different words do not behave the same way.
- For example:
  *the flowers delivered to the patient arrived*

Earley Parsing

- Earley Parsing is a more advanced form of CKY parsing with two novel ideas:
  - A *dotted rule* as a way to get around the explicit conversion of a CFG to Chomsky Normal Form.
  - Do not explore every single element in the CKY parse chart. Instead use goal-directed search.
- Since natural language grammars are quite large, and are often modified to be able to parse more data, avoiding the explicit conversion to CNF is an advantage.
- A dotted rule denotes that the right hand side of a CF rule has been partially recognized/parsed.
- By avoiding the explicit $n^3$ loop of CKY, we can parse some grammars more efficiently, in time $n^2$ or $n$.
- Goal-directed search can be done in any order including left to right (more psychologically plausible).
Earley Parsing

- $S \rightarrow \bullet NP \ VP$ indicates that once we find an $NP$ and a $VP$ we have recognized an $S$
- $S \rightarrow NP \bullet VP$ indicates that we’ve recognized an $NP$ and we need a $VP$
- $S \rightarrow NP \ VP \bullet$ indicates that we have a complete $S$
- Consider the dotted rule $S \rightarrow \bullet NP \ VP$ and assume our CFG contains a rule $NP \rightarrow John$
  Because we have such an $NP$ rule we can predict a new dotted rule $NP \rightarrow \bullet John$

Earley Parsing

- If we have the dotted rule: $NP \rightarrow \bullet John$ and the next input symbol on our input tape is the word $John$ we can scan the input and create a new dotted rule $NP \rightarrow John \bullet$
- Consider the dotted rule $S \rightarrow \bullet NP \ VP$ and $NP \rightarrow John \bullet$
  Since $NP$ has been completely recognized we can complete $S \rightarrow NP \bullet VP$
- These three steps: predictor, scanner and completer form the Earley parsing algorithm and can be used to parse using any CFG without conversion to CNF
  Note that we have not accounted for $\epsilon$ in the scanner
Earley Parsing

- A state is a dotted rule plus a span over the input string, e.g. 
  \((S \rightarrow NP \bullet VP, [4, 8])\) implies that we have recognized an \(NP\)

- We store all the states in a chart – in chart\([j]\) we store all
  states of the form: \((A \rightarrow \alpha \bullet \beta, [i,j])\), where \(\alpha, \beta \in (N \cup T)^*\)

Earley Parsing

- Note that \((S \rightarrow NP \bullet VP, [0, 8])\) implies that in the chart
  there are two states \((NP \rightarrow \alpha \bullet, [0, 8])\) and
  \((S \rightarrow \bullet NP VP, [0, 0])\) — this is the completer rule, the heart
  of the Earley parser

- Also if we have state \((S \rightarrow \bullet NP VP, [0, 0])\) in the chart, then
  we always predict the state \((NP \rightarrow \bullet \alpha, [0, 0])\) for all rules
  \(NP \rightarrow \alpha\) in the grammar
Earley Parsing

\[
S \rightarrow NP \ VP \\
NP \rightarrow Det \ N \mid NP \ PP \mid John \\
Det \rightarrow the \\
N \rightarrow cookie \mid table \\
VP \rightarrow VP \ PP \mid V \ NP \mid V \\
V \rightarrow ate \\
PP \rightarrow P \ NP \\
P \rightarrow on
\]

Consider the input: 0 John 1 ate 2 on 3 the 4 table 5
What can we predict from the state \((S \rightarrow \bullet \ NP \ VP, [0,0])\)?
What can we complete from the state \((V \rightarrow ate \bullet, [1,2])\)?

Earley Parsing

- enqueue(state, j):
  - input: state = \((A \rightarrow \alpha \bullet \beta, [i,j])\)
  - input: j (insert state into chart[j])
  - if state not in chart[j] then
    - chart[j].add(state)
  - end if
- predictor(state):
  - input: state = \((A \rightarrow B \bullet C, [i,j])\)
  - for all rules \(C \rightarrow \alpha\) in the grammar do
    - newstate = \((C \rightarrow \bullet \alpha, [j,j])\)
    - enqueue(newstate, j)
  - end for
Earley Parsing

- scanner(state, tokens):
  - **input:** state = \( (A \rightarrow B \cdot a C, [i, j]) \)
  - **input:** tokens (list of input tokens to the parser)
  - if tokens\([j] == a \) then
    - newstate = \( (A \rightarrow B a \cdot C, [i, j + 1]) \)
    - enqueue(newstate, j+1)
  - end if

- completer(state):
  - **input:** state = \( (A \rightarrow B C \cdot, [j, k]) \)
  - for all rules \( X \rightarrow Y \cdot A Z, [i, j] \in \text{chart}[j] \) do
    - newstate = \( (X \rightarrow Y A \cdot Z, [i, k]) \)
    - enqueue(newstate, k)
  - end for

Earley Parsing

- earley(tokens[0 \ldots N], grammar):
  - for each rule \( S \rightarrow \alpha \) where \( S \) is the start symbol do
    - add \( (S \rightarrow \cdot \alpha, [0, 0]) \) to chart[0]
  - end for
  - for \( 0 \leq j \leq N + 1 \) do
    - for state in chart[j] that has not been marked do
      - mark state
      - if state = \( (A \rightarrow \alpha \cdot B \beta, [i, j]) \) then
        - predictor(state)
      - else if state = \( (A \rightarrow \alpha \cdot b \beta, [i, j]), j < N + 1 \) then
        - scanner(state, tokens)
      - else
        - completer(state)
      - end if
    - end for
  - end for
  - return yes if chart[N + 1] has a final state
Earley Parsing

- **isIncomplete(state):**
  
  ```
  if state is of type (A → α •, [i, j]) then
    return False
  end if
  return True
  ```

- **nextCategory(state):**
  
  ```
  if state == (A → B • ν C, [i, j]) then
    return ν (ν can be terminal or non-terminal)
  else
    raise error
  end if
  ```

Earley Parsing

- **isFinal(state):**
  
  ```
  input: state = (A → α •, [i, j])
  cond1 = A is a start symbol
  cond2 = isIncomplete(state) is False
  cond3 = j is equal to length(tokens)
  if cond1 and cond2 and cond3 then
    return True
  end if
  return False
  ```

- **isToken(category):**
  
  ```
  if category is terminal symbol then
    return True
  end if
  return False
  ```