Today’s Main Points

- Refresher for dynamic programming
  How this applies to parsing.
- Two dynamic programming parsers:
  - CYK
  - Earley’s algorithm
- Hand out Programming Assignment #1
  “Implement CYK”

Programming languages

```c
printf ("/charset %s", (re_opcode_t) *(p - 1) == charset_not ? "^" : "");
assert (p + *p < pend);
for (c = 0; c < 256; c++)
  if (c / 8 < *p && (p[1 + (c/8)] & (1 << (c % 8)))) {
    /* Are we starting a range? */
    if (last + 1 == c && !inrange) {
      putchar ('-');
      inrange = 1;
    } /* Have we broken a range? */
    else if (last + 1 != c && inrange) {
      putchar (last);
      inrange = 0;
    }
    if (!inrange)
      putchar (c);
    last = c;
  }
```

- Easy to parse.
- Designed that way!

Natural languages

- No {} () [] to indicate scope & precedence
- Lots of overloading (arity varies)
- Grammar isn’t known in advance!
- Context-free grammar not best formalism

The parsing problem

```
Grammar

correct test trees

accuracy

```

Applications of parsing (1/2)

  English
  ![map operations](image)
  Chinese
- Speech synthesis from parses (Prevost 1996)
  The government plans to raise income tax.
The government plans to raise income tax the imagination.
- Speech recognition using parsing (Chelba et al 1998)
  Put the file in the folder.
  Put the file and the folder.
Applications of parsing (2/2)

- Grammar checking (Microsoft)
- Indexing for information retrieval (Woods 1997) — washing a car with a hose in vehicle maintenance
- Information extraction (Hobbs 1996) (Miller et al 2000)

Parsing State of the Art

- Recent parsers quite accurate, e.g.,
  - A Maximum-Entropy-Inspired Parser
    Eugene Charniak
  - Three Generative, Lexicalised Models for Statistical Parsing
    Michael Collins

- Most sentences parsed correctly, or with one error

Last class...

- We defined a CFG, where it sits in the Chomsky hierarchy
- Talked about parsing as search...
  through an exponential number of possible trees
- Gave examples of bottom-up and top-down search.
- Discussed problems:
  - Infinite loop with left-recursive rules
  - Much duplicated work in exponential space... backtracking

Dynamic Programming

- (Not much to do with “programming” in the CS sense.)
- Dynamic programming is efficient in finding optimal solutions for cases with lots of overlapping subproblems.
- It solves problems by recombinging solutions to subproblems, when the subproblems themselves may share sub-subproblems.

DP Example: Calculating Fibonacci Numbers

- F(n) = F(n-1) + F(n-2), where F(0)=0, F(1)=1.
  E.g. 0, 1, 1, 2, 3, 5, 8, 13, 21

- Non-DP implementation

function fib(n)
  if n is 0 or 1, return n,
  otherwise, return fib(n-1) + fib(n-2)
end function

DP implementation

function fib(n)
  array f[n];
  f[0] = 0;
  f[1] = 1;
  for i = 2 to requested number do
    f[i] = f[i-1] + f[i-2];
  end function
### Dynamic Programming for Parsing

- Given CFG in Chomsky Normal Form, and an input string, we want to search for valid parse trees.
- What are the intermediate sub-problems?
- What would the dynamic programming table look like?

### CKY algorithm, recognizer version

- **Input**: string of n words
- **Output**: yes/no (since it’s only a recognizer)
- **Data structure**: n x n table
  - rows labeled 0 to n-1
  - columns labeled 1 to n
  - cell [i,j] lists possible constituents spanning words between i and j

#### Time: 1 flies 2 like 3 an 4 arrow 5

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NP</td>
<td>VP</td>
<td>VP</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P</td>
<td>V</td>
<td>S</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Det</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Time: 1 flies 2 like 3 an 4 arrow 5

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NP</td>
<td>VP</td>
<td>VP</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P</td>
<td>V</td>
<td>S</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Det</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
time 1 flies 2 like 3 an 4 arrow 5

1 S → NP VP
6 S → Vst NP
2 S → S PP
1 VP → V NP
2 VP → VP PP
1 NP → Det N
2 NP → NP PP
3 NP → NP NP
0 PP → P NP
time 1 flies 2 like 3 an 4 arrow 5

1 S → NP VP
6 S → Vst NP
2 S → S PP
1 VP → V NP
2 VP → VP PP
1 NP → Det N
2 NP → NP PP
3 NP → NP NP
0 PP → P NP

time 1 flies 2 like 3 an 4 arrow 5

1 S → NP VP
6 S → Vst NP
2 S → S PP
1 VP → V NP
2 VP → VP PP
1 NP → Det N
2 NP → NP PP
3 NP → NP NP
0 PP → P NP
Dynamic Programming Parsing 2

• How about a dynamic programming solution for **arbitrary CGF grammars**?
  
• (Grammars not in Chomsky Normal Form.)

---

Earley Parser (1970)

• Nice combination of
  – dynamic programming
  – incremental interpretation
  – avoids infinite loops
  – no restrictions on the form of the context-free grammar.
  – \( A \rightarrow B \ C \ the \ D \ of \) causes no problems
  – \( O(n^3) \) worst case, but faster for many grammars
  – Uses left context and optionally right context to constrain search.

---

Overview of the Algorithm

• **Finds constituents and partial constituents in input**
  – \( A \rightarrow B \ C \ . \ D \ E \) is partial: only the first half of the \( A \)

---

Earley Parser

• Input: String of words and grammar
  
• Output: yes/no
  (i.e. recognizer, but can turn into a parser)

• Data Structure:
  – columns 0 through \( n \),
  corresponding to the gaps between words
  – column \( j \) is a list of entries like
    \( (i, A \rightarrow X \ Y \ . \ Z \ W) \)
  meaning there could be an \( A \) starting at \( i \), and we have found the \( X \ Y \) part of it from \( i \) to \( j \).

---

Overview of the Algorithm

• Proceeds incrementally left-to-right
  – Before it reads word 5, it has already built all hypotheses that are consistent with first 4 words
  – Reads word 5 & attaches it to immediately preceding hypotheses.
  Might yield new constituents that are then attached to hypotheses immediately preceding them …
  
  • E.g., attaching \( D \) to \( A \rightarrow B \ C \ D \ E \) gives \( A \rightarrow B \ C \ D \ . \ E \)
  • Attaching \( E \) to that gives \( A \rightarrow B \ C \ D \ . \ E \)
  • Now we have a complete \( A \) that we can attach to hypotheses immediately preceding the \( A \), etc.
The Parse Table

- Columns 0 through n corresponding to the gaps between words
- Entries in column 5 look like (3, NP → NP, PP)
  (but we’ll omit the → to save space)
  
  - Built while processing word 5
  - Means that the input substring from 3 to 5 matches the initial NP portion of a NP → NP, PP rule
  - Dot shows how much we’ve matched as of column 5
  - Perfectly fine to have entries like (3, VP → is it, true that S)

What will it mean that we have this entry?

- Unknown right context: Doesn’t mean we’ll necessarily be able to find a VP starting at column 5 to complete the S.
- Known left context: Does mean that some dotted rule back in column 3 is looking for an S that starts at 3.
  
  - So if we actually do find a VP starting at column 5, allowing us to complete the S, then we’ll be able to attach the S to something.
  - And when that something is complete, it too will have a customer to its left...
  
  - In short, a top-down (i.e., goal-directed) parser: it chooses to start building a constituent not because of the input but because that’s what the left context needs. In the spoon, won’t build spoon as a verb because there’s no way to use a verb there.
  
  - So any hypothesis in column 5 could get used in the correct parse, if words 1-5 are continued in just the right way by words 6-n.

Earley’s Algorithm, recognizer version

- Add ROOT → S to column 0.
- For each j from 0 to n:
  - For each dotted rule in column j, (including those we add as we go!)
    - Look at what’s after the dot:
      - If it’s a word w, SCAN:
        - If w matches the input word between j and j+1, advance the dot and add the resulting rule to column j+1
      - If it’s a non-terminal X, PREDICT:
        - Add all rules for X to the bottom of column j, with the dot at the start: e.g. X → Y Z
      - If there’s nothing after the dot, ATTACH:
        - We’ve finished some constituent, A, that started in column j. So for each rule in column j that has A after the dot: Advance the dot and add the result to the bottom of column j.
  - Output “yes” just if last column has ROOT → S.
  
  - NOTE: Don’t add an entry to a column if it’s already there!

Summary of the Algorithm

- Process all hypotheses one at a time in order. (Current hypothesis is shown in blue.)
  
  - This may add new hypotheses to the end of the to-do list, or try to add old hypotheses again.
  
  - Process a hypothesis according to what follows the dot:
    - If a word, scan input and see if it matches
    - If a nonterminal, predict ways to match it
      - (We’ll predict blindly, but could reduce # of predictions by looking ahead k symbols in the input and only making predictions that are compatible with this limited right context)
    - If nothing, then we have a complete constituent, so attach it to all its customers

A Grammar

<table>
<thead>
<tr>
<th>S</th>
<th>NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP →</td>
<td>Det N</td>
</tr>
<tr>
<td>NP →</td>
<td>NP PP</td>
</tr>
<tr>
<td>VP →</td>
<td>V NP</td>
</tr>
<tr>
<td>VP →</td>
<td>VP PP</td>
</tr>
<tr>
<td>PP →</td>
<td>P NP</td>
</tr>
</tbody>
</table>

An Input Sentence

Papa ate the caviar with a spoon.
predict the kind of S we are looking for

Remember this stands for (0, S → NP VP)

predict the kind of NP we are looking for
(actually we'll look for 3 kinds: any of the 3 will do)

predict the kind of NP we are looking for
(2 kinds)

predict the kind of Det we are looking for
(2 kinds)

predict the kind of NP we're looking for
but we were already looking for these so don't add duplicate goals! Note that this happened when we were processing a left-recursive rule.

scan: the desired word is in the input!

scan: failure
attach the newly created NP (which starts at 0) to its customers (incomplete constituents that end at 0 and have NP after the dot)

**scan: failure**

predict

predict

predict
9/19/04

[Diagram]

scan: success!

[Diagram]

scan: failure

[Diagram]

predict (these next few steps should look familiar)
attach (again!)
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 V. ate</td>
<td>2 Det the</td>
<td>3 N caviar</td>
<td>4 P with</td>
<td>5 Det a</td>
<td>6 N caviar</td>
<td>7 N spoon</td>
</tr>
<tr>
<td>1 V. ate</td>
<td>2 Det the</td>
<td>3 N caviar</td>
<td>4 P with</td>
<td>5 Det a</td>
<td>6 N caviar</td>
<td>7 N spoon</td>
</tr>
<tr>
<td>1 V. ate</td>
<td>2 Det the</td>
<td>3 N caviar</td>
<td>4 P with</td>
<td>5 Det a</td>
<td>6 N caviar</td>
<td>7 N spoon</td>
</tr>
<tr>
<td>1 V. ate</td>
<td>2 Det the</td>
<td>3 N caviar</td>
<td>4 P with</td>
<td>5 Det a</td>
<td>6 N caviar</td>
<td>7 N spoon</td>
</tr>
<tr>
<td>6 Papa</td>
<td>1 ate</td>
<td>2 the</td>
<td>3 caviar</td>
<td>4 with a spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>----------</td>
<td>---------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ROOT .</td>
<td>0 NP Papa</td>
<td>1 V ate</td>
<td>2 Det the</td>
<td>3 N caviar</td>
<td>4 with a spoon</td>
<td></td>
</tr>
<tr>
<td>0 S NP VP</td>
<td>0 S NP VP</td>
<td>1 VP V . NP</td>
<td>2 Det the</td>
<td>3  N caviar</td>
<td>4 with a spoon</td>
<td></td>
</tr>
<tr>
<td>0 NP Det N</td>
<td>0 NP Det N</td>
<td>1  N caviar</td>
<td>2 VP V . NP</td>
<td>3  N caviar</td>
<td>4 with a spoon</td>
<td></td>
</tr>
<tr>
<td>0 NP . NP PP</td>
<td>0 NP . NP PP</td>
<td>1 VP V . NP</td>
<td>2 NP Det N</td>
<td>3  N caviar</td>
<td>4 with a spoon</td>
<td></td>
</tr>
<tr>
<td>0 NP Papa</td>
<td>1 VP V . NP</td>
<td>2 NP Det N</td>
<td>3  N caviar</td>
<td>4 with a spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Det . the</td>
<td>1 PP . P NP</td>
<td>2 Det the</td>
<td>1 VP VP PP</td>
<td>2 NP Det N</td>
<td>3  N caviar</td>
<td>4 with a spoon</td>
</tr>
<tr>
<td>0 Det . a</td>
<td>1 V . ate</td>
<td>2 Det . a</td>
<td>4 PP . P NP</td>
<td>4 with a spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 P . with</td>
<td>0 ROOT S</td>
<td>1 P . with</td>
<td>0 ROOT S</td>
<td>1 P . with</td>
<td>0 ROOT S</td>
<td>1 P . with</td>
</tr>
</tbody>
</table>
Left Recursion Kills Pure Top-Down Parsing ...

VP

Left Recursion Kills Pure Top-Down Parsing ...

VP
   VP
   VP
   VP
   PP

Left Recursion Kills Pure Top-Down Parsing ...

VP
   VP
   VP
   VP
   PP
   PP

Left Recursion Kills Pure Top-Down Parsing ...

VP
   VP
   VP
   VP
   PP
   PP
   PP
   PP

makes new hypotheses ad infinitum before we've seen the PPs at all
hypotheses try to predict in advance how many PPs will arrive in input

... but Earley's Alg is Okay!

VP
   VP
   VP
   VP
   PP
   PP
   PP
   PP

1 VP → . VP PP
(in column 1)

... but Earley's Alg is Okay!

VP
   VP
   VP
   VP
   PP
   PP
   PP
   PP

1 VP → . VP PP
(in column 1)

VP
   VP
   VP
   V
   NP

ate the caviar
(in column 4)
... but Earley's Alg is Okay!

VP
1 VP → . VP PP
VP PP
(in column 1)

attach
VP
1 VP → VP . PP
VP PP
V NP
ate the caviar
(in column 4)

VP
1 VP → VP PP
VP PP
with a spoon
V NP
ate the caviar
(in column 7)

... but Earley's Alg is Okay!

VP
1 VP → . VP PP
VP PP
(can be reused)
(in column 1)

VP
1 VP → VP PP
VP PP
with a spoon
V NP
ate the caviar
(in column 7)

... but Earley's Alg is Okay!

VP
1 VP → . VP PP
VP PP
(can be reused again)
(in column 1)

VP
1 VP → VP PP
VP PP
in his bed
V NP
ate the caviar
(in column 10)
... but Earley's Alg is Okay!

\[ \text{VP} \rightarrow 1 \text{VP} \rightarrow \text{VP PP} \]
\[ \text{VP PP} \text{ (in column 1)} \]
\[ \text{VP PP can be reused again} \]
\[ \text{VP PP (in his bed)} \]
\[ \text{VP PP with a spoon} \]
\[ \text{ate the caviar} \]
\[ \text{(in column 10)} \]

---

### How to change the parser into a recognizer?

<table>
<thead>
<tr>
<th>8</th>
<th>Papa</th>
<th>ate</th>
<th>the</th>
<th>caviar</th>
<th>with a spoon</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT  S</td>
<td>0 NP. Papa</td>
<td>1 V. ate</td>
<td>2 Det the.</td>
<td>3 N. caviar</td>
<td>6 N. spoon.</td>
</tr>
<tr>
<td>0 S. Det N</td>
<td>0 NP. VP</td>
<td>1 VP. V. NP</td>
<td>2 VP. Det N</td>
<td>3 N. caviar</td>
<td>5 NP. Det N.</td>
</tr>
<tr>
<td>0 NP. Det N</td>
<td>0 NP. NP. PP</td>
<td>2 NP. Det N</td>
<td>3 N. caviar</td>
<td>1 VP. V. NP</td>
<td>4 PP. P NP.</td>
</tr>
<tr>
<td>0 NP. Det N</td>
<td>0 NP. NP. PP</td>
<td>2 NP. Det N</td>
<td>3 N. caviar</td>
<td>1 VP. V. NP</td>
<td>4 PP. P NP.</td>
</tr>
<tr>
<td>0 NP. Det N</td>
<td>0 NP. NP. PP</td>
<td>2 NP. Det N</td>
<td>3 N. caviar</td>
<td>1 VP. V. NP</td>
<td>4 PP. P NP.</td>
</tr>
</tbody>
</table>

- completed a VP in col 4
- col 1 lets us use it in a VP PP structure
- completed that VP = VP PP in col 7
- col 1 would let us use it in a VP PP structure
- can reuse col 1 as often as we need

---

### What's the Complexity?

- How many state sets will there be?
  - Length of sentence, \( n \)
- How big can the state sets get?
  - Size of grammar, \( G \), times \( n \)
- How long does it take to build a state set?
  - Scan
    - Constant time
  - Predict
    - Need to check for duplicates
    - Complete
    - Search previous state set, also check for duplicates, \( (Gn)^2 \)
  - Total: \( O(n^3) \)