Parsing CFGs

• Consider the problem of parsing with arbitrary CFGs
• For any input string, the parser has to produce a parse tree
• The simpler problem: print yes if the input string is generated by the grammar, print no otherwise
• This problem is called recognition
CKY Recognition Algorithm

- The Cocke-Kasami-Younger algorithm
- As we shall see it runs in time that is polynomial in the size of the input
- It takes space polynomial in the size of the input
- **Remarkable fact:** it can find all possible parse trees (exponentially many) in polynomial time

Chomsky Normal Form

- Before we can see how CKY works, we need to convert the input CFG into Chomsky Normal Form
- CNF means that the input CFG G is converted to a new CFG G’ in which all rules are of the form:
  \[ A \rightarrow BC \]
  \[ A \rightarrow a \]
Epsilon Removal

• First step, remove epsilon rules
  \[ A \rightarrow B \ C \]
  \[ C \rightarrow \varepsilon \mid C \ D \mid a \]
  \[ D \rightarrow b \quad B \rightarrow b \]

• After \( \varepsilon \)-removal:
  \[ A \rightarrow B \mid B \ C \ D \mid B \ a \mid BC \]
  \[ C \rightarrow D \mid C \ D \ D \mid a \ D \mid C \ D \mid a \]
  \[ D \rightarrow b \quad B \rightarrow b \]

Removal of Chain Rules

• Second step, remove chain rules
  \[ A \rightarrow B \ C \mid C \ D \ C \]
  \[ C \rightarrow D \mid a \]
  \[ D \rightarrow d \quad B \rightarrow b \]

• After removal of chain rules:
  \[ A \rightarrow B \ a \mid B \ D \mid a \ D \ a \mid a \ D \ D \mid D \ D \ a \mid D \ D \ D \]
  \[ D \rightarrow d \quad B \rightarrow b \]
Eliminate terminals from RHS

• Third step, remove terminals from the rhs of rules
  \[ A \rightarrow B \ a \ C \ d \]
• After removal of terminals from the rhs:
  \[ A \rightarrow B \ N_1 \ C \ N_2 \]
  \[ N_1 \rightarrow a \]
  \[ N_2 \rightarrow d \]

Binarize RHS with Nonterminals

• Fourth step, convert the rhs of each rule to have two non-terminals
  \[ A \rightarrow B \ N_1 \ C \ N_2 \]
  \[ N_1 \rightarrow a \]
  \[ N_2 \rightarrow d \]
• After converting to binary form:
  \[ A \rightarrow B \ N_3 \ N_4 \]
  \[ N_1 \rightarrow a \]
  \[ N_3 \rightarrow N_1 \ N_4 \]
  \[ N_2 \rightarrow d \]
  \[ N_4 \rightarrow C \ N_2 \]
CKY algorithm

• We will consider the working of the algorithm on an example CFG and input string

• Example CFG:
  \[ S \rightarrow A \, X \mid Y \, B \]
  \[ X \rightarrow A \, B \mid B \, A \]
  \[ Y \rightarrow B \, A \]
  \[ A \rightarrow a \]
  \[ B \rightarrow a \]

• Example input string: \( aaa \)

### CKY Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A, B</td>
<td>X, Y</td>
<td>S</td>
<td>S → A_{(0,1)} X_{(1,3)}</td>
</tr>
<tr>
<td></td>
<td>A → a</td>
<td>X → A B</td>
<td>S → Y_{(0,2)} B_{(2,3)}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → a</td>
<td>Y → B A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A, B</td>
<td>X, Y</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A → a</td>
<td>X → A B</td>
<td>Y → B A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
Parse trees

CKY Algorithm

Input string input of size $n$
Create a 2D table chart of size $n^2$

for $i=0$ to $n-1$

\[ \text{chart}[i][i+1] = \text{A if there is a rule A} \rightarrow \text{a and input}[i]=\text{a} \]

for $j=2$ to $N$

for $i=j-2$ downto $0$

for $k=i+1$ to $j-1$

\[ \text{chart}[i][j] = \text{A if there is a rule A} \rightarrow \text{B C and chart}[i][k] = \text{B and chart}[k][j] = \text{C} \]

return yes if chart[0][n] has the start symbol
else return no
CKY algorithm summary

- Parsing arbitrary CFGs
- For the CKY algorithm, the time complexity is $O(|G|^2 n^3)$
- The space requirement is $O(n^2)$
- The CKY algorithm handles arbitrary ambiguous CFGs
- All ambiguous choices are stored in the chart
- For compilers we would like our grammars to be unambiguous

GLR – Generalized LR Parsing

- Works for any CFG (just like CKY algorithm)
  - Masaru Tomita [1986]
- If you have shift/reduce conflict, just clone your stack and shift in one clone, reduce in the other clone
  - proceed in lockstep
  - parser that get into error states die
  - merge parsers that lead to identical reductions (graph structured stack)
- Careful implementation can provide $O(n^3)$ bound
- However for some grammars, parser will still run in time that is exponential in grammar size
Parsing - Summary

• Parsing arbitrary CFGs using the CKY algorithm: $O(n^3)$ time complexity
• Chomsky Normal Form (CNF) provides the $n^3$ time bound
• LR parsers can be extended to Generalized LR parsers to deal with arbitrary CFGs, complexity is still $O(n^3)$

Parsing - Additional Results

• $O(n^2)$ time complexity for linear grammars
  – All rules are of the form $S \rightarrow aSb$ or $S \rightarrow a$
  – Reason for $O(n^2)$ bound is the linear grammar normal form: $A \rightarrow aB$, $A \rightarrow Ba$, $A \rightarrow B$, $A \rightarrow a$
• Left corner parsers
  – extension of top-down parsing to arbitrary CFGs
• Earley’s parsing algorithm
  – $O(n^3)$ worst case time for arbitrary CFGs just like CKY
  – $O(n^3)$ worst case time for unambiguous CFGs
  – $O(n)$ for specific unambiguous grammars
    (e.g. $S \rightarrow aSa \mid bSb \mid \varepsilon$)