3.6 The Cocke-Younger-Kasami parsing algorithm

Recursive descent parsing is a clear and effective method for parsing
LL(1)-grammars; a string of \( n \) symbols can be parsed in \( O(n) \) time.
However, LL(1)-grammars are a fairly limited class; solving the
general parsing problem is not as easy.

In principle the problem could be solved by applying the recursive
descent parsing method, but in practice the large number of
alternatives to be tested becomes a problem (typically \( O(c^n) \) for
some \( c \geq 2 \)).

The Cocke-Younger-Kasami algorithm is a method for parsing strings
in any context-free grammar \( G \). It is based on the general technique
of dynamic programming (tabulating partial solutions). The method
requires \( O(n^2) \) time, where \( n \) is the length of the string to be parsed.

First some grammar transformations are defined.

1. Removing \( \varepsilon \)-productions

Let \( G = (V, \Sigma, P, S) \) be a context-free grammar. The nonterminal
\( A \in V - \Sigma \) is called nullable if \( A \Rightarrow^* \varepsilon \).

Lemma 3.5. Any context free grammar \( G \) can be transformed into an
equivalent grammar \( G' \) where at most the initial symbol is nullable.

Proof. Let \( G = (V, \Sigma, P, S) \). First determine the nullable nonterminals
of \( G \) as follows:

(i) first set

\[
\text{NULL} := \{ A \in V - \Sigma \mid A \Rightarrow^* \varepsilon \text{ is a production of } G \};
\]

(ii) repeat the following expansion of the set \( \text{NULL} \) until it no longer
grows:

\[
\text{NULL} := \text{NULL} \cup \{ A \in V - \Sigma \mid A \Rightarrow B_1 \ldots B_k \text{ is a production of } G, \ B_i \in \text{NULL} \text{ for all } i = 1, \ldots, k \}.
\]
Now replace each production \( A \rightarrow X_1 \ldots X_k \) by the set of productions of the form
\[
A \rightarrow \alpha_1 \ldots \alpha_k, \quad \text{where}
\]
\[
\alpha_i = \begin{cases} 
X_i, & \text{if } X_i \notin \text{NULL}; \\
X_i \text{ or } \varepsilon, & \text{if } X_i \in \text{NULL}.
\end{cases}
\]
Finally remove all productions of the form \( A \rightarrow \varepsilon \). If the production \( S \rightarrow \varepsilon \) would be to remove, we introduce the new initial symbol \( S' \) and the productions \( S' \rightarrow S \) and \( S' \rightarrow \varepsilon \). □

Example. Remove the \( \varepsilon \)-productions from the grammar:
\[
\begin{align*}
S & \rightarrow A \mid B \\
A & \rightarrow aBa \mid \varepsilon \quad \Rightarrow \quad (\text{NULL} = \{A, B, S\}) \\
B & \rightarrow bAb \mid \varepsilon \\
S & \rightarrow A \mid B \mid \varepsilon \\
A & \rightarrow aBa \mid aa \mid \varepsilon \\
B & \rightarrow bAb \mid bb.
\end{align*}
\]

Proof. Let \( G = (V, \Sigma, P, S) \). Find first the “unit successors” of each nonterminal in \( G \) as follows:
(i) at start for each \( A \in V - \Sigma \) let
\[
F(A) := \{ B \in V - \Sigma \mid A \rightarrow B \text{ is a production of } G \};
\]
(ii) repeat the following expansion operation of the \( F \)-sets until they no longer grow:
\[
F(A) := F(A) \cup \bigcup \{ F(B) \mid A \rightarrow B \text{ is a production of } G \}.
\]
After this remove all unit productions from \( G \) and replace them with all possible productions of the form \( A \rightarrow \omega \), where \( B \rightarrow \omega \) is a non-unit production of \( G \) for some \( B \in F(A) \). □

2. Removing unit productions
A production of the form \( A \rightarrow B \), where \( A \) and \( B \) are nonterminals, is a unit production.

Lemma 3.6. Any context-free grammar \( G \) can be transformed into an equivalent grammar \( G' \) without unit productions.
Example. Remove the unit productions from the previous grammar:

\[ S' \rightarrow S | \varepsilon \]
\[ F(S') = \{ S, A, B \}, F(S) = \{ A, B \}, \]
\[ S \rightarrow A \mid B \]
\[ F(A) = F(B) = \{ \varepsilon \}. \]
By replacing the unit productions as described we obtain the grammar:

\[ S' \rightarrow aBa | aa | bAb | bb | \varepsilon \]
\[ S \rightarrow aBa | aa | bAb | bb \]
\[ A \rightarrow aBa \mid aa \]
\[ B \rightarrow bAb \mid bb. \]

(We may note that \( S \) is now “unnecessary”, i.e., it cannot occur in the derivation of any sentence in the grammar. Unnecessary nonterminals can be removed from the grammar by a similar method. Exercise.)

The Chomsky normal form

A context free grammar \( G = (V, \Sigma, P, S) \) is in Chomsky normal form, if no nonterminal other than \( S \) is nullable, and apart from the possible production \( S \rightarrow \varepsilon \) the remaining productions are of the form

\[ A \rightarrow BC \quad \text{or} \quad A \rightarrow a, \]

where \( A, B \) and \( C \) are nonterminals and \( a \) is a terminal.

For simplicity it is additionally required that the initial symbol \( S \) does not appear on the right hand side of any production.

Theorem 3.7. Any context free grammar \( G \) can be transformed into an equivalent grammar \( G' \) that is in Chomsky normal form.

Proof. Let \( G = (V, \Sigma, P, S) \). First remove the \( \varepsilon \)-productions and unit productions from \( G \) by the constructions in Lemmata 3.5 and 3.6. Now all productions in \( G \) are of the form \( A \rightarrow a \) or \( A \rightarrow X_1 \ldots X_k, \)
\[ k \geq 2 \text{ (or } S \rightarrow \varepsilon). \]

For each terminal \( a \) add a new nonterminal \( C_a \) and a production \( C_a \rightarrow a \). Then first replace all terminals in productions of the form \( A \rightarrow X_1 \ldots X_k, k \geq 2 \), with the new nonterminals, and then replace the whole production by the set of productions

\[ A \rightarrow X_1 A_1 \]
\[ A_1 \rightarrow X_2 A_2 \]
\[ \vdots \]
\[ A_{k-2} \rightarrow X_{k-1} X_k, \]

where \( A_1, \ldots, A_{k-2} \) are again new nonterminals. \( \square \)
The Chomsky normal form obtained by the previous construction:

\[ S \rightarrow CD_1 \]
\[ S_1 \rightarrow BCD_1 \]
\[ S_2 \rightarrow CD \]
\[ S \rightarrow CD_2 \]
\[ S_1 \rightarrow CD_1 \]
\[ B \rightarrow b \]
\[ C \rightarrow c \]

Example. A grammar:
\[ S \rightarrow aBCd \mid bbb \]
\[ B \rightarrow b \]
\[ C \rightarrow c \]
\[ C_a \rightarrow a \]
\[ C_b \rightarrow b \]
\[ C_c \rightarrow c \]
\[ C_d \rightarrow d. \]

The CYK algorithm

Let \( G = (V, \Sigma, P, S) \) be a context-free grammar. By Theorem 3.7 we may assume that \( G \) is in Chomsky normal form. The question whether the string \( x \) is in the language \( L(G) \) can be solved as follows:

If \( x = \varepsilon \), then \( x \in L(G) \) iff \( S \rightarrow \varepsilon \) is a production of \( G \).

Otherwise denote \( x = a_1 \ldots a_n \) and investigate producing various substrings of \( x \).

Let \( N_k \) denote the set of those nonterminals \( A \) from which one can derive the substring of \( x \) that starts at position \( i \) and whose length is \( k \):

\[ N_k = \{ A \in V - \Sigma \mid A \vdash_G^* a_i \ldots a_{i+k-1}, \ 1 \leq i \leq i+k-1 \leq n. \} \]

The sets \( N_k \) can be computed by tabulating them from shorter to longer strings as presented in the following. Clearly \( x \in L(G) \) iff \( S \in N_{|x|} \).

Example. A grammar \( G \) in Chomsky normal form:

\[ S \rightarrow AB \mid BC \]
\[ A \rightarrow BA \mid a \]
\[ B \rightarrow CC \mid b \]
\[ C \rightarrow AB \mid a \]

The computation of the CYK algorithm with grammar \( G \) and input \( x = baaba \):

\[ S \rightarrow AB \]
\[ A \rightarrow BA \]
\[ B \rightarrow CC \]
\[ C \rightarrow AB \]

<table>
<thead>
<tr>
<th>( N_k )</th>
<th>1:b</th>
<th>2:a</th>
<th>3:a</th>
<th>4:b</th>
<th>5:a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>A.C</td>
<td>A.C</td>
<td>B</td>
<td>A.C</td>
</tr>
<tr>
<td>2</td>
<td>S.A</td>
<td>B</td>
<td>B</td>
<td>S.A</td>
<td>-</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>S.A</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>S.A</td>
<td>C</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>S.A</td>
<td>C</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Since the initial symbol \( S \) is in \( N_5 \), we deduce that \( x \) is in \( L(G) \).
In computing a set $N_k$ the CYK algorithm proceeds simultaneously towards the set $N_k$ in column $N_i$ and away from $N_k$ along the diagonal $N_{i+j,k+j}$:

\[ N_k \]

### Definition 3.2
A pushdown automaton is a 6-tuple 

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, F) \]

where
- $Q$ is the finite set of states;
- $\Sigma$ is the input alphabet;
- $\Gamma$ is the stack alphabet;
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow P(Q \times (\Gamma \cup \{\varepsilon\}))$ is the (set-valued) transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of (accepting) final states.

The interpretation of the value

\[ \delta(q, \sigma, \gamma) = \{(q_1, \gamma_1), \ldots, (q_k, \gamma_k)\} \]

of the transition function is that in state $q$ upon reading the input symbol $\sigma$ and the stack symbol $\gamma$ the automaton may move to one of the states $q_1, \ldots, q_k$ and replace the top element of the stack by one of the symbols $\gamma_1, \ldots, \gamma_k$ respectively. In the general case, pushdown automata are therefore nondeterministic.

If $\sigma = \varepsilon$, the automaton makes a transition without reading an input symbol. If $\gamma = \varepsilon$, the automaton does not read a stack symbol and the new written symbol is put on the top of the stack (a push operation). If the symbol read from the stack is $\gamma \neq \varepsilon$ and the symbol to be written is $\gamma' = \varepsilon$, the element on the top of the stack is removed (a pop operation).

A pushdown automaton is like a finite state automaton, to which a stack of unbounded size has been added. Using the stack is fairly limited: the automaton may read, write, remove or add symbols only at the top of the stack.
The configuration of the automaton is the triple \((q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*\); in particular, the initial configuration with input \(x\) is the triple \((q_0, x, \varepsilon)\).

Intuition: in configuration \((q, w, \alpha)\) the automaton is in state \(q\), the remaining part of the input string is \(w\) the stack, read from top to bottom, contains the string \(\alpha\).

The configuration \((q, w, \alpha)\) directly leads to configuration \((q', w', \alpha')\), denoted by

\[
(q, w, \alpha) \xrightarrow{\sigma} (q', w', \alpha'),
\]

if we may write \(w = \sigma w', \alpha = \gamma \beta, \alpha' = \gamma' \beta' (|\sigma|, |\gamma|, |\gamma'| \leq 1)\), such that

\[
(q', \gamma') \in \delta(q, \sigma, \gamma).
\]

Example. ⬤ A pushdown automaton for the language \(\{a^k b^k \mid k \geq 0\}\):

\[M = \{(q_0, q_1, q_2, q_3), \{a, b\}, \{A, \varepsilon\}, \delta, q_0, \{q_2, q_3\}\},\]

missä

\[
\begin{align*}
\delta(q_0, a, \varepsilon) &= \{(q_1, A)\}, \\
\delta(q_1, a, \varepsilon) &= \{(q_1, A)\}, \\
\delta(q_1, b, A) &= \{(q_2, \varepsilon)\}, \\
\delta(q_1, b, A) &= \{(q_3, \varepsilon)\}, \\
\delta(q_2, b, A) &= \{(q_2, \varepsilon)\}, \\
\delta(q_2, b, A) &= \{(q_3, \varepsilon)\}, \\
\delta(q, \sigma, \gamma) &= \emptyset \quad \text{for other } (q, \sigma, \gamma).
\end{align*}
\]

A graph presentation:

\[
\begin{tikzpicture}
  \node (q0) at (0,0) {q_0};
  \node (q1) at (1,1) {q_1};
  \node (q2) at (1,-1) {q_2};
  \draw[->] (q0) edge [loop above] node {a, A\varepsilon} (q0);
  \draw[->] (q0) edge node {a, A\varepsilon} (q1);
  \draw[->] (q1) edge node {b, A\varepsilon} (q2);
  \draw[->] (q1) edge node {b, A\varepsilon} (q2);
  \draw[->] (q2) edge [loop below] node {b, A\varepsilon} (q2);
\end{tikzpicture}
\]

The computation of the automaton with input \(aabb\):

\[
(q_0, aabb, \varepsilon) \xrightarrow{a} (q_1, abb, A) \xrightarrow{b} (q_2, b, A) \xrightarrow{b} (q_3, \varepsilon).
\]

Since \(q_3 \in F = \{q_0, q_3\}\), we have \(aabb \in L(M)\).
Pushdown automata and context-free languages

Theorem 3.8 A language is context-free, if and only if it can be recognized by some (non-deterministic) pushdown automaton. □

The proof is omitted here, but the principle in constructing the pushdown automaton M_G that corresponds to the given grammar G is:
1. The stack of the automaton is used to simulate the left derivation
2. If the top element of the stack is a non-terminal, some production of G is applied and the corresponding symbols pushed onto the stack;
3. If the topmost element is a terminal, it is matched with the next input symbol.

Example. The pushdown automaton that corresponds to the grammar \( \{ S \rightarrow aSbS \mid bSaS \} \):

\[
\begin{align*}
S & \Rightarrow aSbS \\
aSbS & \Rightarrow abSaSbS \\
& \Rightarrow abaSbS \\
& \Rightarrow abab
\end{align*}
\]

For example there is an accepting computation for the input \( abab \):

\[
(q_0, abab, \varepsilon) \xrightarrow{(q, bab, S\#)} (q, bab, aSbS\#) \xrightarrow{(q, ab, SaSbS\#)} (q, ab, aSbS\#) \xrightarrow{(q, b, Sb\#)} (q, b, bS\#) \xrightarrow{(q, \varepsilon, \varepsilon)} (q, \varepsilon, \varepsilon)
\]

This corresponds to the left derivation of abab in the grammar:

\[
S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow abab.
\]

A pushdown automaton \( M \) is deterministic, if every configuration \((q, w, \alpha)\) has at most one possible immediate successor \((q', w', \alpha')\), for which

\[
(q, w, \alpha) \xrightarrow{(q', w', \alpha')}
\]

Unlike for finite state automata, nondeterministic pushdown automata are strictly more powerful than deterministic ones. For example the language \( \{ w^R \mid w \in \{a, b\}^* \} \) can be recognized by a nondeterministic, but not by a deterministic, pushdown automaton (proof omitted).

A context-free language is deterministic, if it can be recognized by a deterministic pushdown automaton. Deterministic languages can be parsed in \( O(n) \) time; general context-free languages require nearly \( O(n^2) \) time with known methods.