CONTEXT FREE LANGUAGES, PARSING AND PARSE CHARTS

Context free languages

An alphabet \( \Sigma \) is a (finite) set of indivisible symbols.
Ex: \( \Sigma = \{0, 1\} \) or \( \Sigma = \{\text{the}, \text{dog}\} \).

A sentence \( S \) (over an alphabet \( \Sigma \)) is a finite string of terminals drawn from \( \Sigma \): \( S \in \Sigma^* \).
Ex: \( S = 011 \).

A language \( L \) is a (possibly infinite) set of sentences.
Ex: \( L = \{01, 011, 0111, 01111, \ldots\} \).

A context free grammar (CFG) is a tuple \( G = (\Sigma, N, R, P) \):

- \( \Sigma \) is an alphabet of terminal symbols
- \( N \) is an alphabet of non-terminal symbols
  \( V = \Sigma + N \) is the vocabulary
- \( R \in N \) is a special root (or start) symbol
- \( P \) is a finite set of productions (or rules) of the form \( \alpha \Rightarrow \beta \) where \( \alpha \in N \) and \( \beta \in V^* \).
  \( \alpha \) is known as the left hand side (lhs) and \( \beta \) the right hand side (rhs) of the production.\(^1\)

The language \( L(G) \) generated by a context free grammar \( G = (\Sigma, N, R, P) \) is the (possibly infinite) set of sentences produced by the following non-deterministic process:

1. Let string \( T = R \)
2. If \( T \) contains no non-terminals, halt and return \( T \)
3. Let \( X \) be the leftmost occurrence of a non-terminal in \( T \); replace \( X \) with the rhs \( \beta \) of any production \( X \Rightarrow \beta \in P \) (if no such production exists, fail)
4. Goto 2

A language \( L \) is context free if there is a CFG \( G \) s.t. \( L = L(G) \).

A (leftmost) derivation (or parse) of a sentence \( S \) under grammar \( G \) is a sequence of productions that successively expand \( R \) to \( S \) in the above process. If multiple leftmost derivations exist for some sentence \( S \) in \( L(G) \), \( S \) is ambiguous under \( G \):

\(^1\)More general phrase structure grammars permit the lhs \( \alpha \) to be more complicated than a single non-terminal; they are much more powerful but can not be processed efficiently.
Example: Dyck languages

Let $G = (\Sigma, N, R, P)$ where $\Sigma = \{ (, [, ] \}$, $N = \{ X \}$, $R = X$, and $P$ contains the productions

$$
X \Rightarrow XX \\
X \Rightarrow (X) \\
X \Rightarrow [X] \\
X \Rightarrow 
$$

Then $L(G)$ is the set of matched-parenthesis strings over two types of parentheses, also known as the semi-Dyck set $D_2$.

A derivation of $(([]))$ under $G$ is

<table>
<thead>
<tr>
<th>Production</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>$X$</td>
</tr>
<tr>
<td>$X \Rightarrow (X)$</td>
<td>$(X)$</td>
</tr>
<tr>
<td>$X \Rightarrow XX$</td>
<td>$(XX)$</td>
</tr>
<tr>
<td>$X \Rightarrow [X]$</td>
<td>$(X)X$</td>
</tr>
<tr>
<td>$X \Rightarrow (}$</td>
<td>$(X)$</td>
</tr>
<tr>
<td>$X \Rightarrow (X)$</td>
<td>$(X)$</td>
</tr>
<tr>
<td>$X \Rightarrow (}$</td>
<td>$(X)$</td>
</tr>
</tbody>
</table>

Derivations can be graphically interpreted as parse trees:

```
    R
     |
    -- x ---
   /  |  \
  |   |   |
  |   |   |
 ( -- x -- -- x -- )
```

Note that $([[]])$ is unambiguous, but $([[]])$ is ambiguous - the sequence of three inner brackets can be derived as a left-branching or right-branching tree. Some languages are inherently ambiguous: any grammar for the language has multiple derivations for some sentences; $D_2$ is not - a more carefully written grammar would not allow for multiple derivations.
Regular languages

A regular language \( L \) is one for which \( L = L(G) \) where \( G \) is a CFG restricted to productions where the rhs \( \beta \) is restricted to be either empty or a terminal followed by a non-terminal. Regular languages are equivalently those that can be recognized by a regular expression or finite state machine.

Ex:

\[
\begin{align*}
A & \Rightarrow 1 \ B \\
A & \Rightarrow \\
B & \Rightarrow 0 \ B \\
B & \Rightarrow 0 \ A
\end{align*}
\]

generates the regular language \((10^+)\)^*. The non-terminal plays the role of the state in a finite state machine.

Recognition, parsing and parse charts

A recognizer is a process that takes a grammar \( G \) and a sentence \( S \) and determines whether \( S \in L(G) \).

A parser is a process that takes a grammar \( G \) and a sentence \( S \) and determines whether \( S \in L(G) \), and if so, returns one or more derivations of the sentence.

The simplest efficient parsing and recognition algorithm for CFGs is the CKY algorithm\(^2\) (figure 1). CKY works bottom-up, building a truth table \( T[i,j,x] \) of whether the non-terminal \( x \) can generate the subsequence \( S_{ij} \).

CKY presupposes grammars have been altered to a simple binary-branching form called Chomsky Normal Form (CNF) where each rhs \( \beta \) is restricted to either a terminal or a pair of nonterminals \((\beta \in \Sigma \text{ or } \beta \in \mathcal{N}N)\).\(^3\) Any CFG \( G \) can be translated to a CNF grammar \( G' \) s.t. \( L(G) = L(G') \).

CKY recognition time is cubic in sentence length \( n \) (notice the triple-nested loop: for every one of the quadratic number of subsequences \( i \ldots j \), must consider a linear number of split points \( k \)). The most widely used recognition and parsing algorithms are \( O(n^3g^2) \) where \( g \) is the number of nonterminals in the grammar, but unlike the naïve CKY algorithm perform much better than worst-case for specialized grammars, (e.g., linear in the case of regular grammars).

The space of possible derivations can be constructed by modifying the CKY recognizer into a parser (figure 2) that builds a chart, which is a directed acyclic graph with AND and OR interior nodes.

For example, parsing the sentence I saw dogs on Mars with the following grammar (in CNF)

\(^2\)Cocke, Kasimi, Younger
\(^3\)The special rule \( R \Rightarrow \) is also allowed, to account for languages containing the empty string.
CKY_recognize(G = <Sigma,N,R,P>, S = S1..Sn)
Initialize T[i,j,x] = False
For i from 1 to n
    If X -> Si in P
        T[i,i,X] = True
For i from n-1 downto 1
    For j from i+1 to n
        For X -> Y Z in P
            For k from i to j-1
                If T[i,k,Y] and T[k+1,j,Z]
                    T[i,j,X] = True
        Return T[1,n,R]

Figure 1: The CKY recognizer

CKY_parse(G = <Sigma,N,R,P>, S = S1..Sn)
Initialize T[i,j,x] = nil
For i from 1 to n
    If X -> Si in P
        T[i,i,X] = new Term(Si)
For i from n-1 downto 1
    For j from i+1 to n
        For X in N
            Let D = []
            For X -> Y Z in P
                For k from i to j-1
                    Let node1 = T[i,k,Y]
                    Let node2 = T[k+1,j,Z]
                    If node1 != nil & node2 != nil
                        D += new AND(X -> Y Z, node1, node2)
            If length(D) == 1
                T[i,j,X] = D_1
            Else if length(D) => 1
                T[i,j,X] = new OR(D_1, ..., D_m)
        Return T[1,n,R]

Figure 2: The CKY parser
\[ S \Rightarrow NP \ VP \\
NP \Rightarrow I \\
NP \Rightarrow dogs \\
NP \Rightarrow Mars \\
NP \Rightarrow NP \ PP \\
PP \Rightarrow P \ NP \\
VP \Rightarrow V \ NP \\
VP \Rightarrow VP \ PP \\
P \Rightarrow on \\
V \Rightarrow saw \]

produces the following chart (in order of node creation)

\[
T[1, 1, NP] = \text{Term}(I) \\
T[2, 2, V] = \text{Term}(saw) \\
T[3, 3, NP] = \text{Term(dogs)} \\
T[4, 4, P] = \text{Term(on)} \\
T[5, 5, NP] = \text{Term(Mars)} \\
T[4, 5, PP] = \text{AND}(PP \Rightarrow P \ NP, T[4, 4, P], T[5, 5, NP]) \\
T[3, 5, NP] = \text{AND}(NP \Rightarrow NP \ PP, T[3, 3, NP], T[4, 5, PP]) \\
T[2, 3, VP] = \text{AND}(VP \Rightarrow V \ NP, T[2, 2, V], T[3, 3, NP]) \\
T[2, 5, VP] = \text{OR}(\text{AND}(VP \Rightarrow V \ NP, T[2, 2, V], T[3, 5, NP]), \text{AND}(VP \Rightarrow VP \ PP, T[2, 3, VP], T[4, 5, PP])) \\
T[1, 5, S] = \text{AND}(S \Rightarrow NP \ VP, T[1, 1, NP], T[2, 5, VP])
\]

which graphically looks like

\[
\text{Term(‘I’)} \quad \text{Term(‘saw’)} \quad \text{Term(‘dogs’)} \quad \text{Term(‘on’)} \quad \text{Term(‘Mars’)}
\]

The size of the chart is \(O(n^3)\) but can represent an exponential number of derivations for highly ambiguous grammars.

To generate a particular derivation from a chart, descend from the root. At an AND, collect the production and the results of recursively descending each daughter. At an OR, non-
deterministically choose one daughter and recursively descend. Different parses result from different choices at OR nodes.

A useful way of thinking of a node of a parse chart is as a compact representation of a set of (sub-)sentences, or equivalently, a set of (sub-)derivations. Thus, one can define the (finite) language of a node:

\[
\begin{align*}
L(\text{Term}(c)) &= \{t\} \\
L(\text{OR}(d_1, d_2)) &= L(d_1) \cup L(d_2) \\
L(\text{AND}(d_1, d_2)) &= \{\text{cat}(s_1, s_2)|s_1 \in L(d_1), s_2 \in L(d_2)\}
\end{align*}
\]

**Recognition as intersection and input lattices**

An alternative view of recognition is to treat the sentence \(S\) as a language with one element \(L(S) = \{S\}\) and the recognition problem as that of determining whether \(L(S) \cap L(G) \neq \emptyset\). This view is nice because it naturally extends to recognition or parsing of more interesting classes of languages than singleton sentences. In particular, given a regular language \(L(G_{reg})\) and a context free language \(L(G_{cfg})\) it is possible to efficiently determine whether \(L(G_{reg}) \cap L(G_{cfg}) \neq \emptyset\) using a slight variation of the CKY algorithm where the table entries \(T[i,j,x]\) are indexed by pairs of non-terminals of \(G_{reg}\) (states in a FSA). That is, in a recognizer \(T[i,j,x]\) is true if the non-terminal \(x\) from \(G_{cfg}\) can expand to a sub-sentence which is can also be produced by the finite state machine described by \(G_{reg}\) starting at state \(i\) and ending at state \(j\).

Then a derivation is a description of how to generate a sentence that is in the intersection of the two languages. Crucially, charts have exactly the same AND/OR structure and parsing complexity is essentially the same: \(O(n^3)\) where \(n\) is the number of different non-terminals (states) in \(G_{reg}\).4

Speech recognizers are often built as a two stage process, where a low-level acoustic/phonological processor generates a lattice of possible phoneme sequences that is represented as a finite state machine or equivalently, regular grammar. This regular grammar is parsed using a more complex context-free grammar describing higher-level phenomena, to produce a chart of all derivations of the CFG that are consistent with any possible phoneme sequence allowed by the acoustic stream.

If a set of local transformations on derivations are defined such as

\[
\begin{align*}
-\text{NP}- & \rightarrow -\text{NP'}- \\
| & | \\
\text{adj} & \text{N} & \text{N'} & \text{adj'} \\
| & | \\
\text{dog} & \text{chien}
\end{align*}
\]

---

4Interestingly, Dyck languages intersected with regular languages are universal in the sense that for any CF language \(L\) there is an integer \(m\) and a regular language \(Q\) s.t. \(L \simeq D_m \cap Q\) where \(\simeq\) is essentially equivalence after local renaming - this is the Chomsky-Schützenberger theorem.
are defined, then a (polynomial-size) parse chart representing an exponential set of derivations can be transformed efficiently to another (polynomial-size) parse chart representing the transformed derivations, preserving all the ambiguity. Thus, in theory one can

- convert a speech signal to a phoneme lattice
- parse the phoneme lattice into a chart
- translate the source-language chart to a target language chart
- generate one or more sentences from the target language chart

all the while retaining ambiguity and polynomial-sized representations. In practice this is usually not practical for natural language grammars of realistic complexity, especially as there are usually non-context-free phenomena one wishes to account for that have to be dealt with differently, and local tree transformations are insufficient to model many cross-language structural divergences.

Weighted grammars and generation

Suppose each production \( p \) is assigned an arbitrary cost or weight \( w(p) \) and the goal is to generate from a chart the \( k \) lowest-weight derivations.

An efficient way to do this (see figure 3) is to first extend the CKY parser to associated with every node the minimum weight of any sub-derivation it represents, and then use the A* algorithm to enumerate, where each state is a tuple (nodes, derivation) where nodes is a set of unexpanded chart nodes and derivation is a partial result - a set of productions. A priority queue over states is maintained, ordered by the minimum weight complete derivation that can result from further expanding state nodes.

Because the weight bounds are computed perfectly by the CKY parser, there is no unnecessary exploration in the search procedure and generation time is essentially linear in the number of derivations \( k \).

In stochastic context free grammars (SCFGs), productions are assigned expansion probabilities, where \( \text{Pr}(X \Rightarrow \beta) \) must be non-negative and normalized such that the sum of \( \text{Pr}(X \Rightarrow \beta) \) over all productions with lhs \( X \) is 1. To convert from such probabilities to weights, the negative logarithm is taken: the resulting weights can be interpreted as energies in a statistical physical model where the joint probability of a derivation with productions \( p_1 \ldots p_n \) is proportional to \( e^{-W} \) where \( W = \sum w(p_i) \).

Cost functions that define the weight of a derivation to be the sum of independent production or terminal weights are called decomposable. Probability distributions derived from decomposable cost functions (namely SCFGs) are the only possible probability distributions over derivations consistent with the context-free assumption of independent expansion of non-terminals. However there are many useful ways to assign weights to the productions or terminals of a grammar that have only strained interpretations in terms of probabilities, for example weights derived from various discriminative training procedures.
CKY_parse(G = <Sigma,N,R,P>, S = S1..Sn)
Initialize T[i,j,x] = nil
For i from 1 to n
  If p = X -> Si in P
    T[i,i,X] = new Term(w(p), Si)
For i from n-1 downto 1
  For j from i+1 to n
    For X in N
      Let D = []
      Let w = inf
      For X -> Y Z in P
        For k from i downto j-1
          Let <node1,w1> = T[i,k,Y]
          Let <node2,w2> = T[k+1,j,Z]
          If node1 != nil & node2 != nil
            D += AND(w1+w2, X -> Y Z, node1, node2)
            w = min(w, w1+w2)
        If length(D) == 1
          T[i,j,X] = D[1]
        Else if length(D) => 1
          T[i,j,X] = new OR(w, D[1], ..., D[m])
      Return T;

A*_enumerate(G = <Sigma,N,R,P>, S = S1..Sn, k) {
  Let T[i,j,x] = CKY_parse(G, S)
  Let results = {}
  Let Root = T[I,n,R]
  If Root != nil
    insert(Q, {[ Root }, {}>, root.weight)
  While (k > 0 & size(Q) > 0)
    <nodes, derivation>, weight = deQueue(Q)
    If (nodes = { })
      results += derivation
      k = k - 1
    Else
      Let [node | rest] = nodes
      If node = AND(w, p, d1, d2)
        insert(Q, <rest + d1 + d2, derivation + p>, weight)
      Else if node = OR(w, d1, ..., dm)
        For d in d1 .. dm
          Let new_weight = weight + (d.weight - w)
          insert(Q, <rest + d, derivation>, new_weight)
      Else % terminal
        insert(Q, <rest, derivation>, weight)
  Return results

Figure 3: A* enumeration