CMPS 260 (Theoretical Foundations of CS)
The CYK Algorithm

The CYK algorithm (named for Cocke, Young, and Kasami, each of whom develeped it independently of the others in the mid-1960's) solves the membership problem for context-free grammars in Chomsky Normal Form. That is, given as input a CFG $G$ in Chomsky Normal Form (CNF) and a string $w$, the algorithm determines whether or not $w \in L(G)$. Because any context-free grammar can be transformed into CNF (with the possible loss of the empty string from the generated language), this gives us a way of solving the membership problem for all CFG's. Letting $G$ and $w$ denote its two inputs (and assuming that CYK is a boolean function that, given a CNF grammar and a string, returns **true** iff the string is generated by the grammar), the algorithm is as follows:

```
function Member_of( G : CFG; w : string ) return boolean is

begin
  if w=e then
     if S is erasable      --(where S is the start symbol of G)
        then return true;
        else return false;
     end if;
  else  -- w /= e
     G' := CNF grammar generating L(G) - {e};
     return CYK(G',w);
  end if;
```

Recall that a context-free grammar is said to be in Chomsky normal form if all its rules are of one of the two forms $A \to b$ or $A \to BC$, where $b$ is a terminal symbol and $B$ and $C$ are nonterminals.

The CYK algorithm is based on the following. Let $G$ be a context-free grammar in Chomsky normal form, and let $w = a_1 a_2 \cdots a_n$ ($a_i \in \Sigma$) be a string over the terminal alphabet $\Sigma$ of $G$. For $i$ and $j$ satisfying $1 \leq i \leq j \leq n$, let $w_{i,j}$ denote the substring $a_i a_{i+1} \cdots a_j$ of $w$ beginning with its $i$-th symbol and ending with its $j$-th symbol, and let $N_{i,j}$ denote the set of nonterminals in $G$ from which $w_{i,j}$ can be derived. That is,

$$N_{i,j} = \{A : A \stackrel{*}{\Rightarrow} w_{i,j}\}$$

**Lemma 1:** For all $i$, $N_{i,i} = \{A : A \to a_i \text{ is a rule in } G\}$

**Proof:** Because $G$ is in CNF, the only way that a string of length one can be derived from a nonterminal symbol is via an application of a rule of the form $A \to b$.

**Lemma 2:** For all $i$ and $j$ satisfying $1 \leq i < j \leq n$, $A \in N_{i,j}$ if and only if there exist nonterminals $B$ and $C$ and a number $k$ satisfying $i \leq k < j$ such that $A \to BC$ is a rule in $G$, $B \in N_{i,k}$, and $C \in N_{k+1,j}$.

**Proof:** Sufficiency (if):

$$
\begin{aligned}
& A \to BC \text{ is a rule } \wedge B \in N_{i,k} \wedge C \in N_{k+1,j} \\
= \quad & < \text{ by defn of } N > \\
& A \to BC \text{ is a rule } \wedge B \stackrel{*}{\Rightarrow} w_{i,k} \wedge C \stackrel{*}{\Rightarrow} w_{k+1,j} \\
\Longrightarrow \quad & < \text{ by properties of derivations } > \\
& A \Rightarrow BC \stackrel{*}{\Rightarrow} w_{i,k}C \stackrel{*}{\Rightarrow} w_{i,k} \cdot w_{k+1,j} \\
\Longrightarrow \quad & < \text{ by properties of derivations } > \\
& A \stackrel{*}{\Rightarrow} w_{i,k} \cdot w_{k+1,j} \\
\Longrightarrow \quad & < w_{i,j} = w_{i,k} \cdot w_{k+1,j} > \\
& A \stackrel{*}{\Rightarrow} w_{i,j} \\
= \quad & < \text{ by defn of } N_{i,j} > \\
& A \in N_{i,j}
\end{aligned}
$$

Necessity (only if):

$$
\begin{aligned}
& A \in N_{i,j} \\
= \quad & < \text{ defn of } N_{i,j} > \\
& A \stackrel{*}{\Rightarrow} w_{i,j} \\
\Longrightarrow \quad & < \text{ see note below } > \\
& A \Rightarrow BC \stackrel{*}{\Rightarrow} w_{i,j} \text{ for some nonterminals } B, C \\
\Longrightarrow \quad & < \text{property of derivations } > \\
& A \to BC \text{ is a rule } \wedge B \stackrel{*}{\Rightarrow} w_{i,k} \wedge C \stackrel{*}{\Rightarrow} w_{k+1,j} \text{ for some } B, C, k \\
= \quad & < \text{ defn of } N_{i,k}, N_{k+1,j} > \\
& A \to BC \text{ is a rule } \wedge B \in N_{i,k} \wedge C \in N_{k+1,j} \text{ for some } B, C, k
\end{aligned}
$$

**Note:** The second step in the proof of necessity is justified by the fact that, in a CNF CFG, a derivation of a terminal string of length two or more from a nonterminal symbol $A$ must begin

with the application of a rule of the form $A \to BC$.
**End of proof of Lemma 2.**

**Lemma 3:** $w \in L(G)$ if and only if $S \in N_{1,n}$, where $S$ is the start symbol of $G$.

**Proof:**

$$
\begin{aligned}
& w \in L(G) \\
= & \quad < \text{ defn of } L(G) > \\
& S \stackrel{*}{\Rightarrow} w \\
= & \quad < w = w_{1,n} > \\
& S \stackrel{*}{\Rightarrow} w_{1,n} \\
= & \quad < \text{ defn of } N_{1,n} > \\
& S \in N_{1,n}
\end{aligned}
$$

From Lemma 3, it follows that, in order to determine whether $w \in L(G)$, it suffices to compute the set $N_{1,n}$ and then to check whether $S$ is a member of that set. But how can we compute $N_{1,n}$? Lemmas 1 and 2 provide strong suggestions. Lemma 1 tells us that, for any $i$, in order to compute $N_{i,i}$ it suffices to examine each rule in $G$. Lemma 2 tells us that, for any $i$ and $m$ such that $1 \leq i < i+m \leq n$, in order to compute $N_{i,i+m}$ it suffices to examine each rule in $G$, as well as the sets $N_{i,k}$ and $N_{k+1,i+m}$ for $k$ satisfying $i \leq k < i+m$. From this we conclude that the "correct" order in which to compute the $N_{i,i+m}$'s is in increasing order of $m$. That way, each time we are to compute a particular set $N_{i,i+m}$, all the sets $N_{i,k}$ and $N_{k+1,i+m}$ $(i \leq k < i+m)$ on which its value depends have been computed already.

We arrive at the following algorithm (on the next page). (For ease of typesetting, in the algorithm we enclose subscripts within square brackets.)

```
CYK Algorithm.

Input:  CFG G in CNF, string w = a[1] a[2] a[3] ... a[n]
Output: YES if w in L(G), NO otherwise


   for i in 1..n  loop
      N[i,i] := empty set;
      for each nonterminal A in G  loop
         if A --> a[i] is a production in G  then    --note that a[i] = w[i,i]
            insert A into N[i,i];
         end if;
      end loop;
   end loop;

   for m in 1..n-1  loop
      --compute N[i,i+m] for i satisfying 1 <= i <= n-m
      for i in 1..n-m  loop
         --compute N[i,i+m]
         N[i,i+m] := empty set;
         for k in i..i+m-1  loop
            for each production A --> BC in G  loop
               if B is in N[i,k]  and  C is in N[k+1,i+m]  then
                  insert A into N[i,i+m];
               end if;
            end loop;
         end loop;
      end loop;
   end loop;

   if S is in N[1,n]  then    --w in L(G) iff S is in N[1,n]
      then  return YES;
      else  return NO;
   end if;
```