# Parsing using CYK Algorithm

- Transform grammar into Chomsky Form:
    1. remove unproductive symbols
    2. remove unreachable symbols
    3. remove epsilons (no non-start nullable symbols)
    4. remove single non-terminal productions X::=Y
    5. transform productions of arity more than two
    6. make terminals occur alone on right-hand side

  Have only rules X ::= Y Z, X ::= t
- Apply CYK dynamic programming algorithm

**Questions:**

- With steps in the order above, what is the worst-case increase in grammar size, in each step and overall?
- Does any step break the effect of a previous one?
- Propose alternative step order and answer again the above.
- Which steps could we omit and still have CYK working?

# Suggested Order

- Removing epsilons (3) can increase grammar size exponentially
- This problem is avoided if we make rules binary first (5).
- Removing epsilons can make some symbols unreachable, so we can repeat 2
- Resulting order:

  1,2,5,3,4,2,6

# A CYK for Any Grammar

grammar G, non-terminals $A_1,\ldots,A_K$, tokens $t_1,\ldots.t_L$

input word: $w = w_{(0)}w_{(1)} \ldots w_{(N-1)}$

$w_{p..q} = w_{(p)}w_{(p+1)} \ldots w_{(q-1)}$

Triple (A, p, q) means: $A \Rightarrow^* w_{p..q}$ , A can be: $A_i$, $t_j$, or $\varepsilon$

**$P = \{(w_{(i)},i,i+1) \mid 0 \leq i < N-1\}$**

**repeat {**

**choose rule $(A ::= B_1 \ldots B_m) \in G$**

**if $((A,p_0,p_m) \notin P$ &&**

**$((m=0 \ \&\& \ p_0=p_m) \ || \ (B_1,p_0,p_1), \ldots,(B_m,p_{m-1},p_m) \in P))$**

**$P := P \cup \{(A,p_0,p_m)\}$**

**} until no more insertions possible**

What is the maximal number of steps?
How long does it take to check step for a rule?

} for grammar in given normal form

# Observation

- How many ways are there to split a string of length Q into m segments?

$$\binom{Q+m}{m} = \frac{(Q+m)!}{Q!\,m!}$$

- Exponential in m, so algorithm is exponential.
- For binary rules, m=2, so algorithm is efficient.

# Name Analysis Problems Detected

- a class is defined more than once: **class A { ...} class B { ... } class A { ... }**
- a variable is defined more than once: **int x; int y; int x;**
- a class member is overloaded (forbidden in Tool, requires **override** keyword in Scala):
  **class A { int x; ... } class B extends A { int x; ... }**
- a method is overloaded (forbidden in Tool, requires **override** keyword in Scala):
  **class A { int x; ... } class B extends A { int x; ... }**
- a method argument is shadowed by a local variable declaration (forbidden in Java, Tool):
  **def (x:Int) { var x : Int; ...}**
- two method arguments have the same name: **def (x:Int,y:Int,x:Int) { ... }**
- a class name is used as a symbol (as parent class or type, for instance) but is not declared:
  **class A extends Objekt {}**
- an identifier is used as a variable but is not declared:
  **def(amount:Int) { total = total + ammount }**
- the inheritance graph has a cycle: **class A extends B {} class B extends C {} class C extends A**

To make it efficient and clean to check for such errors, we associate mapping from each identifier to the symbol that the identifier represents.

- We use Map data structures to maintain this mapping (Map, what else?)
- The rules that specify how declarations are used to construct such maps are given by ***scope* rules of the programming language.**