Bottom-up parsing algorithms

Cocke–Younger–Kasami algorithm

and

Chomsky Normal Form

Last time

Showed how to use JavaCUP for getting ASTs

But we never saw HOW the parser works

This time

Dip our toe into parsing

- Approaches to Parsing
- CFG transformations
  - Useless non-terminals
  - Chomsky Normal Form: A form of grammar that’s easier to deal with
- CYK: powerful, heavyweight approach to parsing

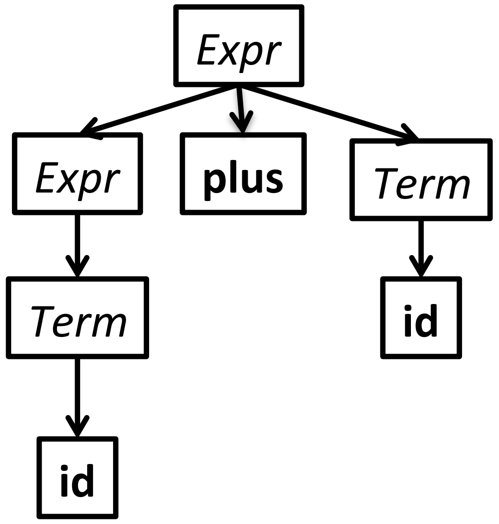
Approaches to parsing

Top Down / "Goal driven"

- Start at root of parse tree, grow downward to match the string

Bottom Up / "Data Driven"

- Start at terminal, generate subtrees until you get to the start



CYK: A general approach to Parsing (*Cocke–Younger–Kasami algorithm*)

- Operates in $O(n^3)$
- Works Bottom-Up
- Only takes a grammar in Chomsky Normal Form
  - This will not turn out to be a limitation

Two parameters:
length of code, size of grammar.
The n here referes to the length of code.

We care about the length of code more,
since the grammar becomes fixed.

Chomsky Normal Form

All rules must be one of two forms:

$X \longrightarrow \mathbf{t}$ (terminal)

$X \longrightarrow A\ B$

The only rule allowed to derive epsilon is the start *S*

nonterm -> term
nonterm -> nonterm nonterm

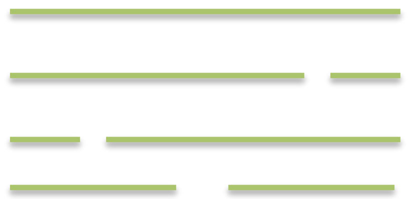
X -> ε is not allowed, unless the LHS is the start symbol: S -> ε. Then S cannot appear on the RHS of any rule.

So to use CYK, you need to rewrite grammar, which means that your parse tree would look differently, though the language would be the same.

What CNF buys CYK

- The fact that non-terminals come in pairs allows you to think of a subtree as a subspan of the input
- The fact that non-terminals are not nullable (except for start) means that each subspan has at least one character

Number of possible splits are O(n).



s = s1 s2 s3 s4

Bottom-up DP: tabularization.

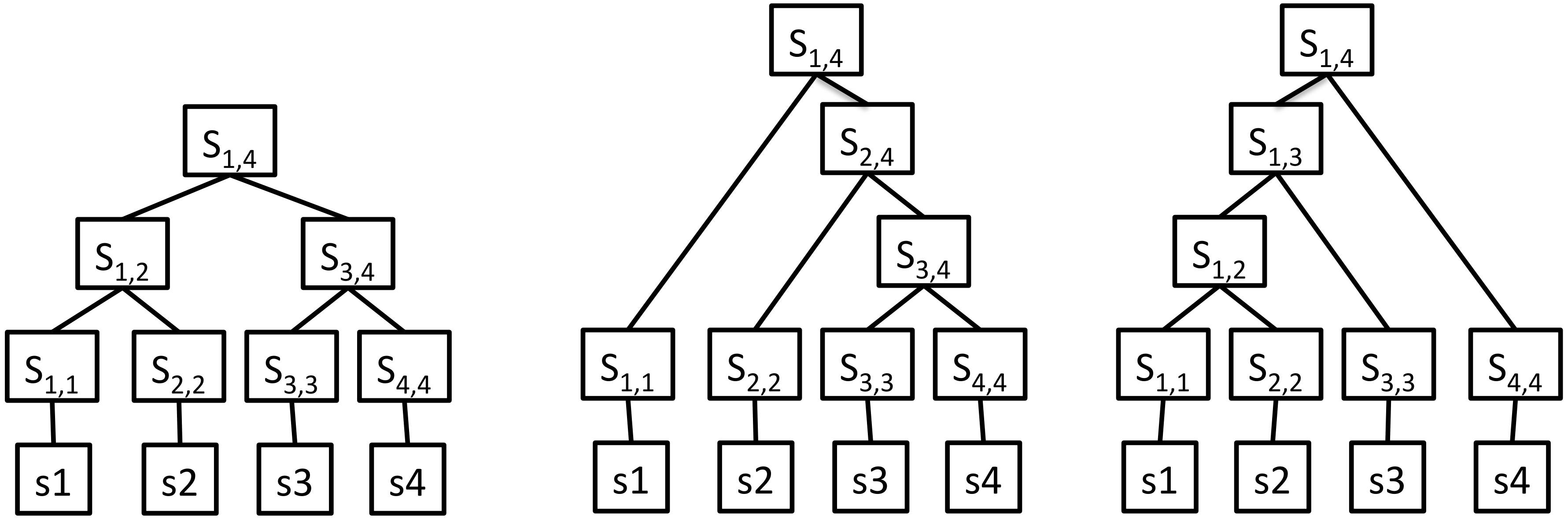
CYK: Dynamic Programming

$X \longrightarrow \mathbf{t}$

Prods. form the leaves of the parse tree

$X \longrightarrow A\ B$

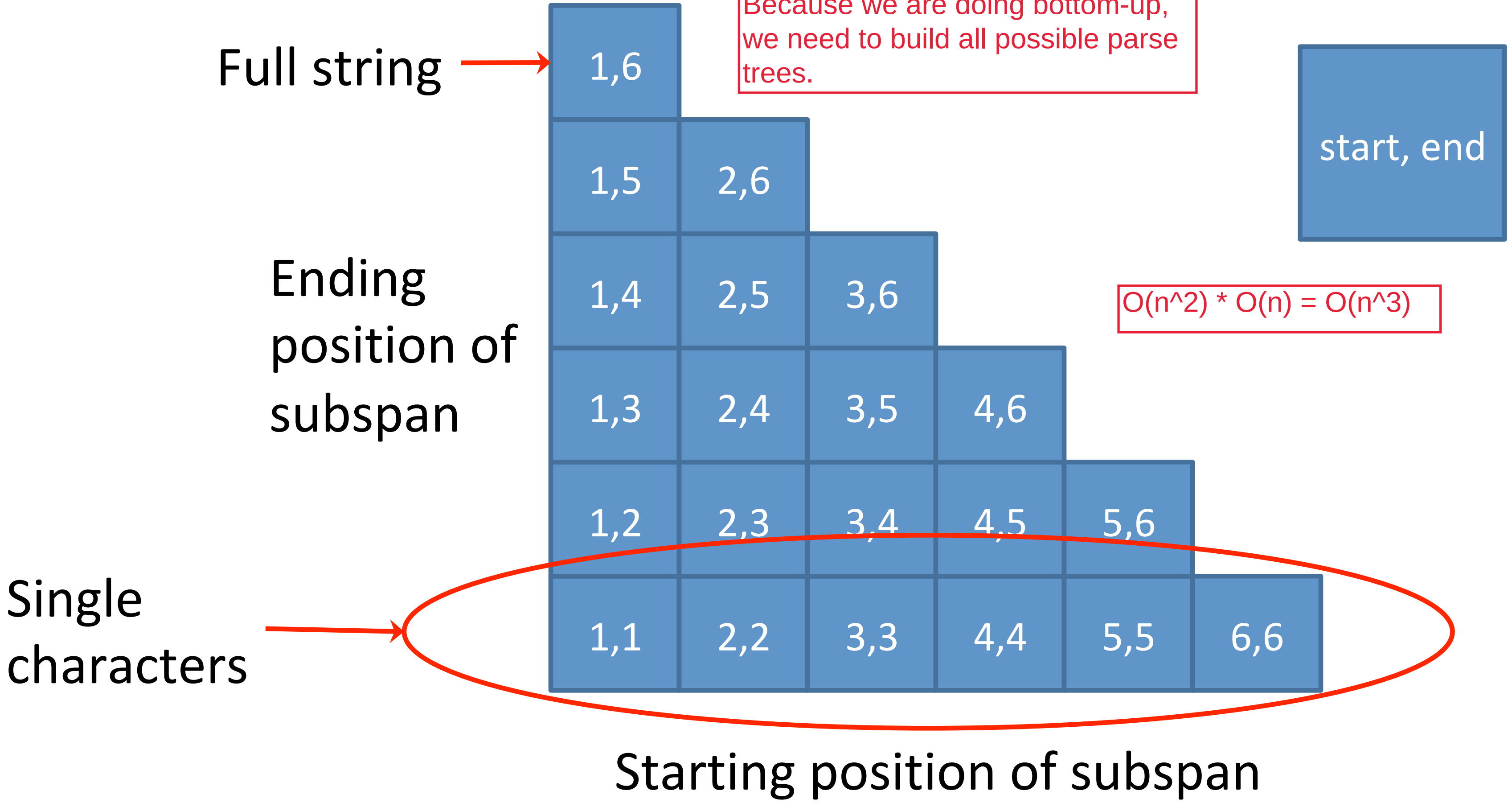
Form binary nodes



Running CYK …

Track every viable subtree from leaf to root. Here are all the subspans for a string of 6 terminals:

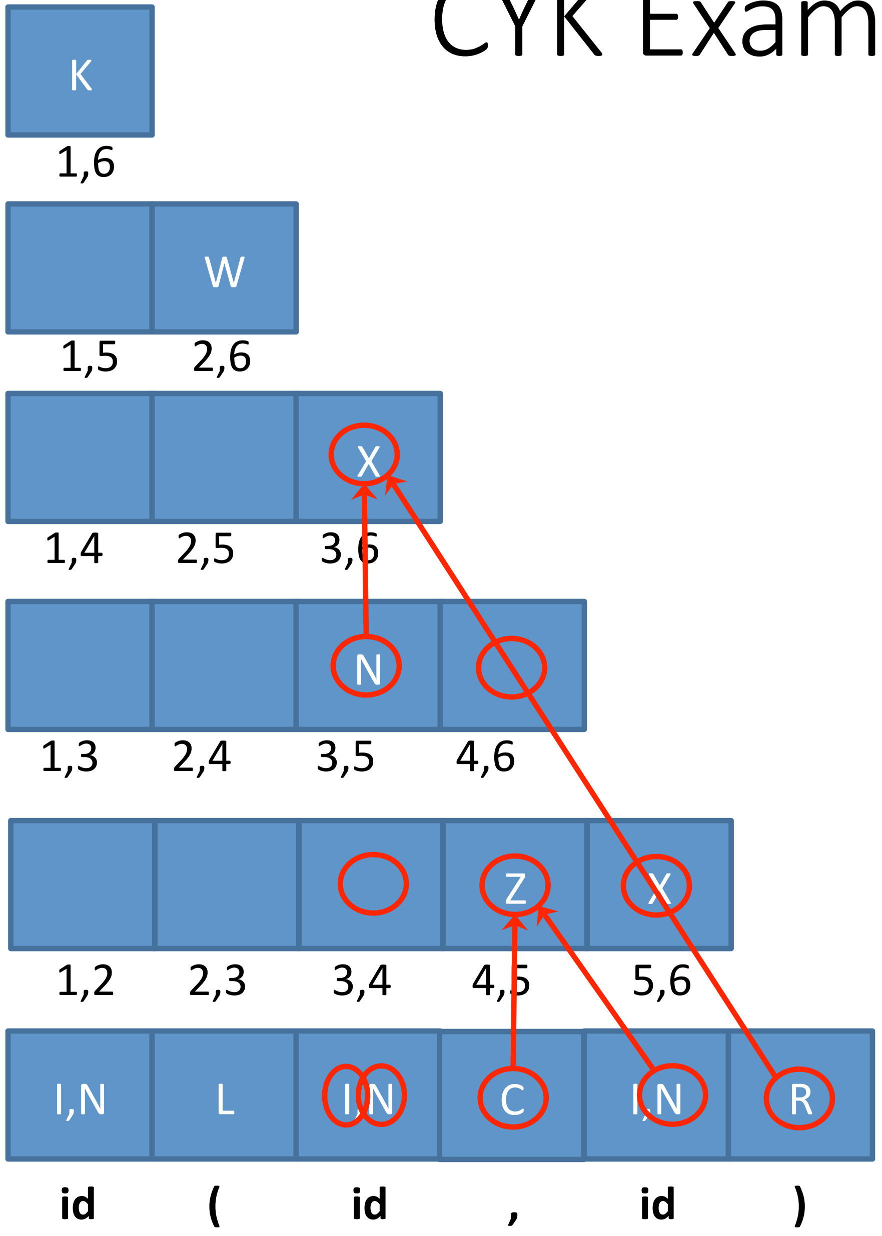
Because we are doing bottom-up, we need to build all possible parse trees.



O(n^2) * O(n) = O(n^3)

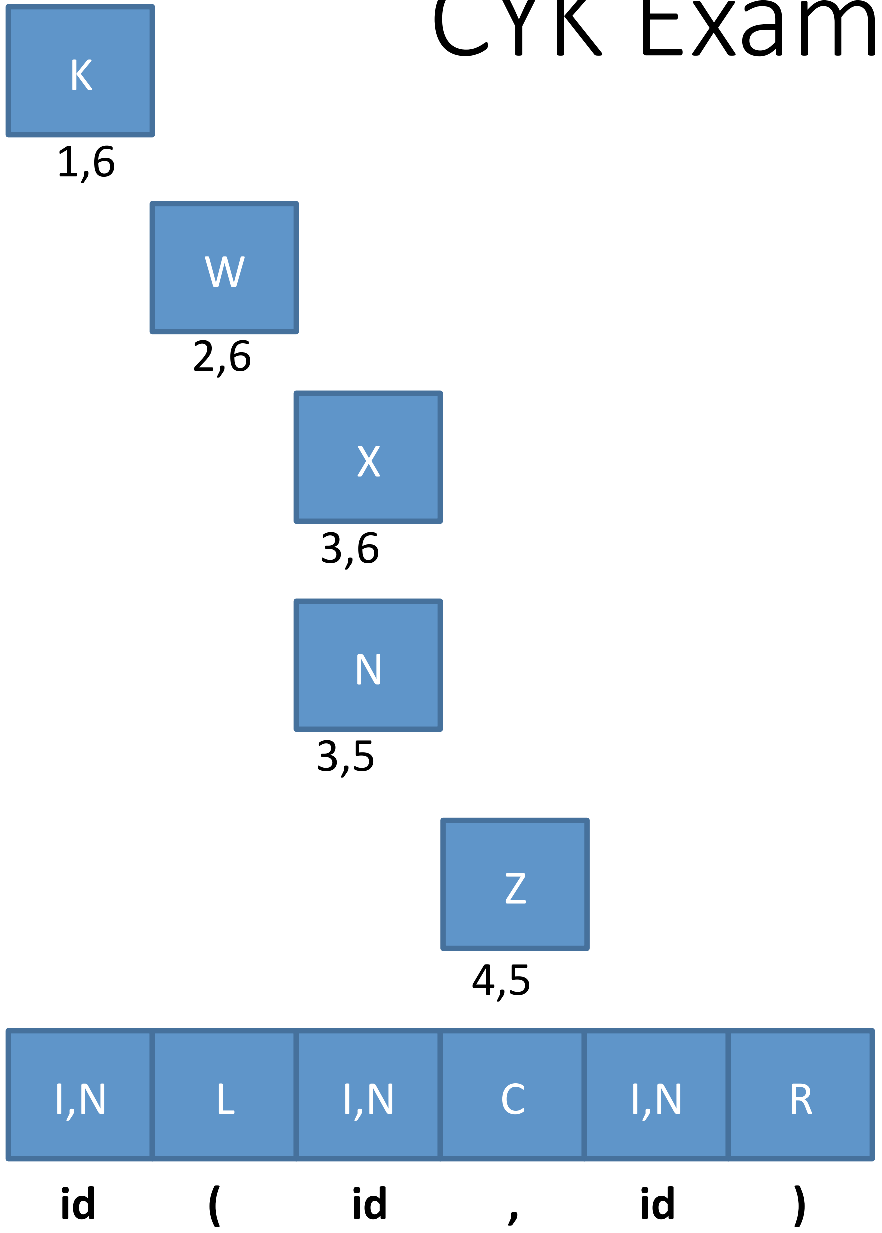
If a string is at the top of the CYK, it is accepted.

CYK Example



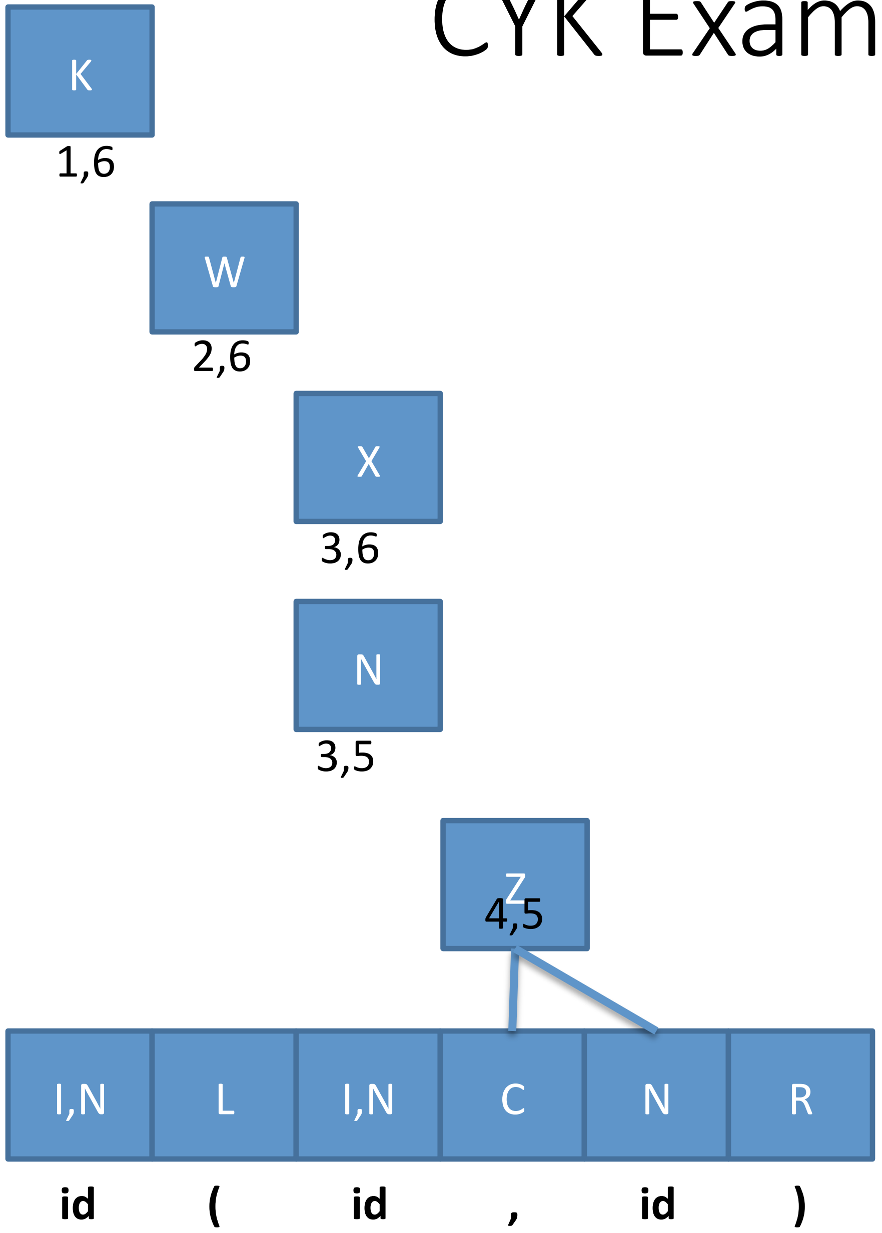
K	⟶	I W
K	⟶	I Y
W	⟶	L X
$\boxed{X}$	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	**(**
R	⟶	**)**
C	⟶	**,**

CYK Example



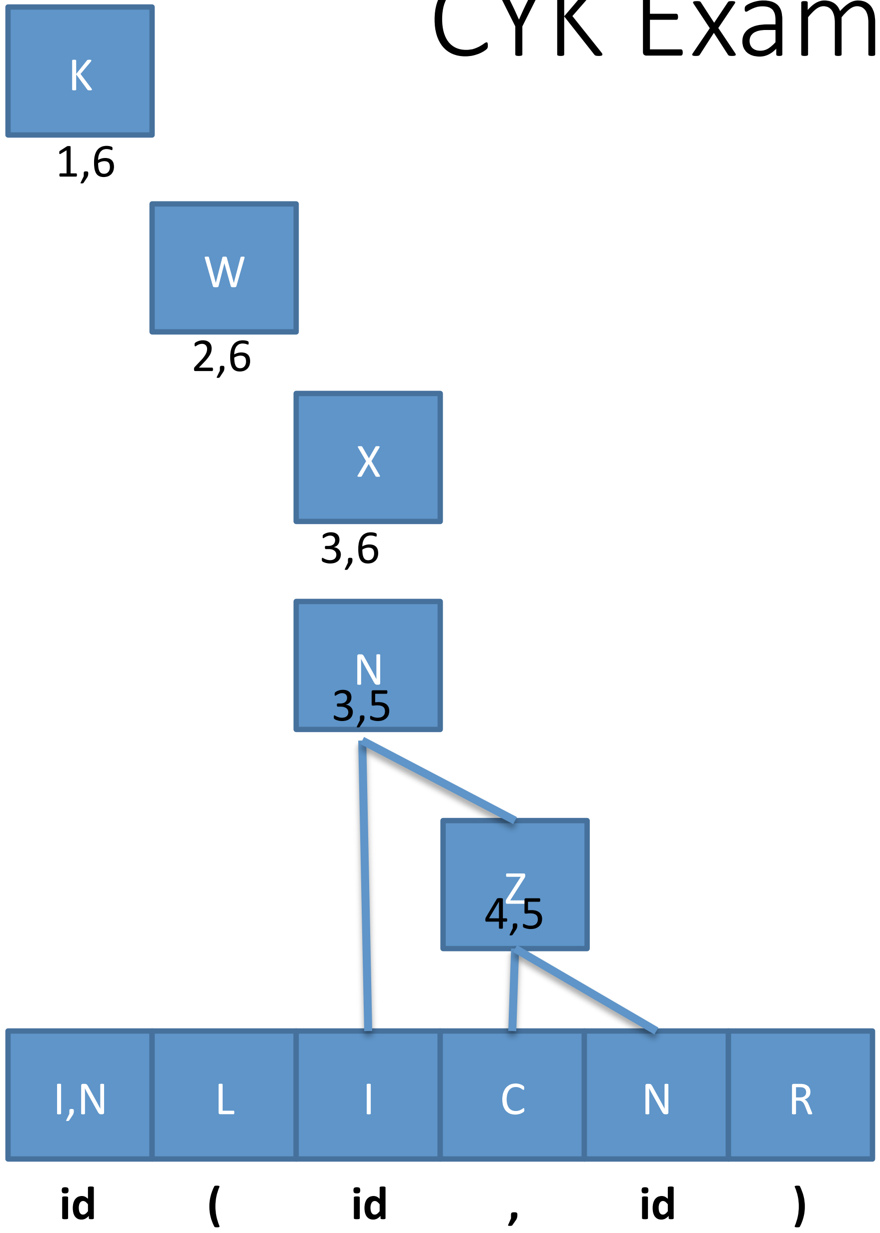
K	$\longrightarrow$	I W
K	$\longrightarrow$	I Y
W	$\longrightarrow$	L X
X	$\longrightarrow$	N R
Y	$\longrightarrow$	L R
N	$\longrightarrow$	**id**
N	$\longrightarrow$	I Z
Z	$\longrightarrow$	C N
I	$\longrightarrow$	**id**
L	$\longrightarrow$	**(**
R	$\longrightarrow$	**)**
C	$\longrightarrow$	**,**

CYK Example



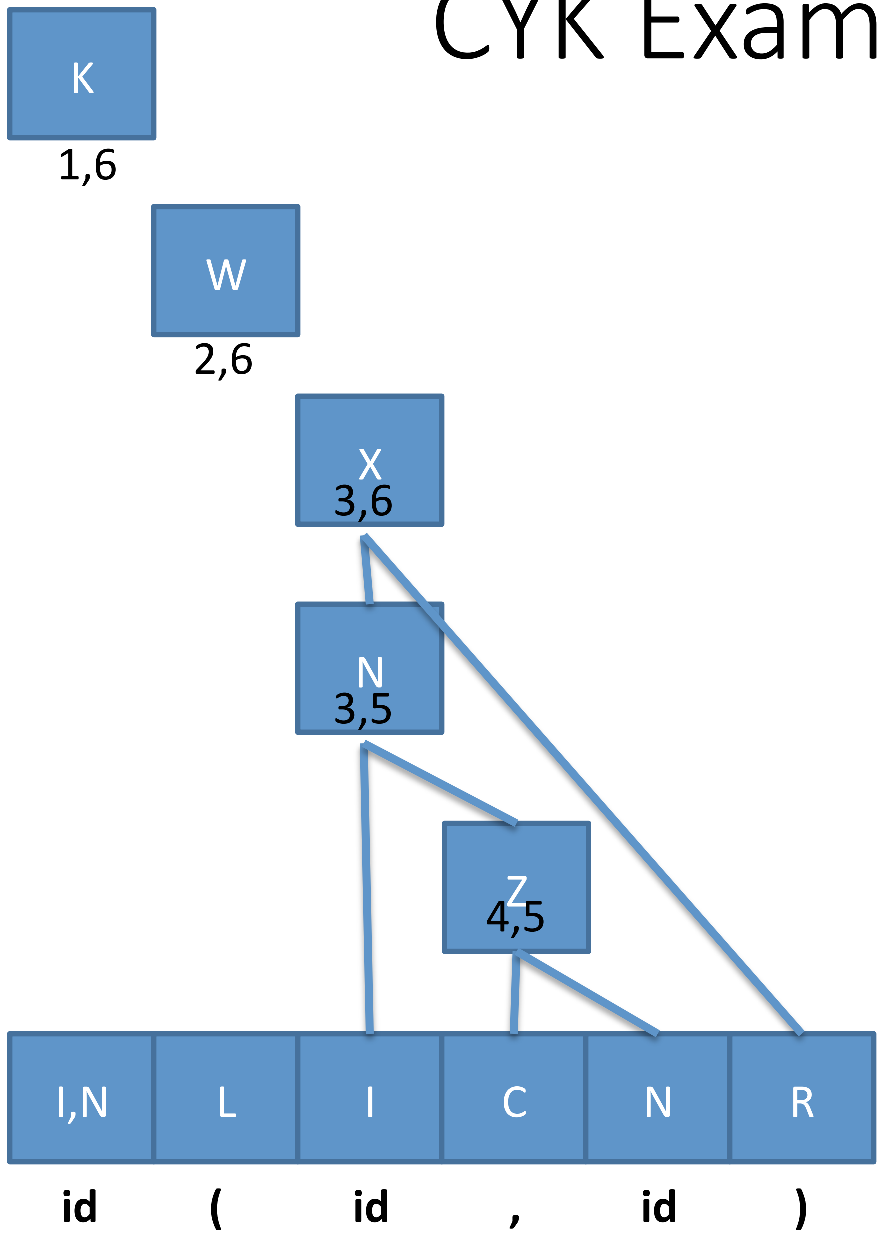
K	→	I W
K	→	I Y
W	→	L X
X	→	N R
Y	→	L R
N	→	**id**
N	→	I Z
Z	→	C N
I	→	**id**
L	→	**(**
R	→	**)**
C	→	**,**

CYK Example



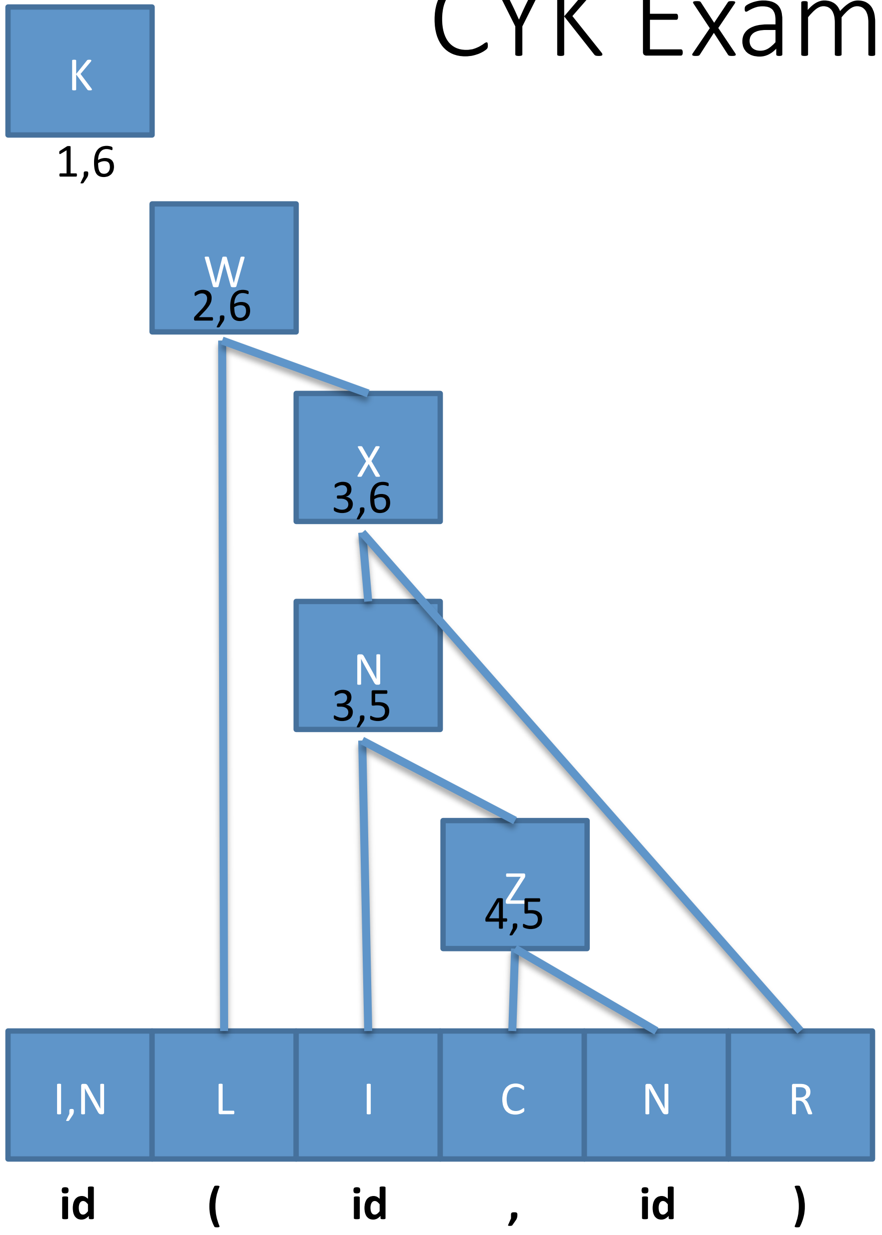
K	→	I W
K	→	I Y
W	→	L X
X	→	N R
Y	→	L R
N	→	**id**
N	→	I Z
Z	→	C N
I	→	**id**
L	→	**(**
R	→	**)**
C	→	**,**

CYK Example



K	⟶	I W
K	⟶	I Y
W	⟶	L X
X	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	(
R	⟶	)
C	⟶	,

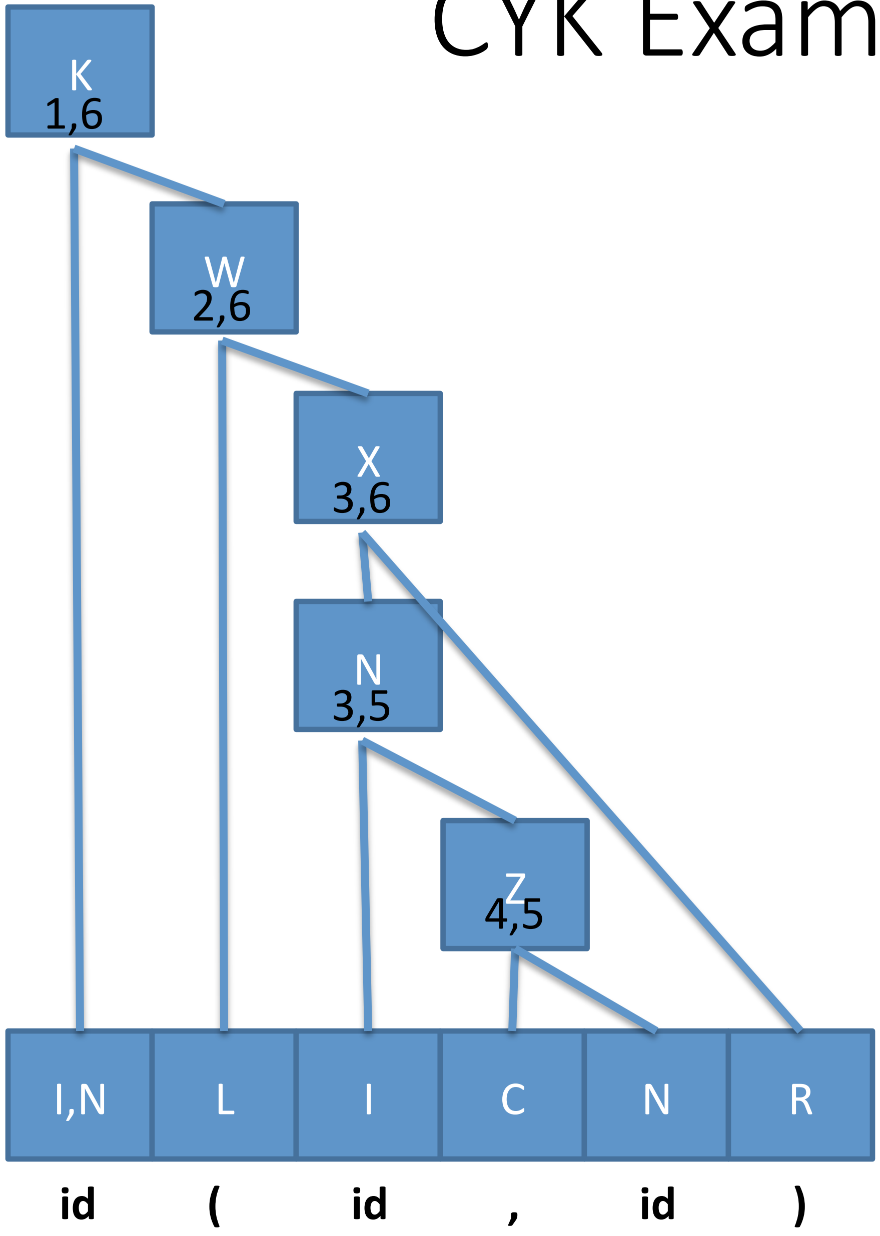
CYK Example



K	⟶	I W
K	⟶	I Y
W	⟶	L X
X	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	**(**
R	⟶	**)**
C	⟶	**,**

If there are more than 1 choices at the top level -> the grammar is ambiguous.

CYK Example



K	⟶	I W
K	⟶	I Y
W	⟶	L X
X	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	**(**
R	⟶	**)**
C	⟶	**,**

Cleaning up our grammars

We want to avoid unnecessary work

- Remove *useless* rules

Eliminating Useless Nonterminals

1. If a nonterminal cannot derive a sequence of terminal symbols then it is useless
2. If a nonterminal cannot be derived from the start symbol, then it is useless

Intuitively, rule 1 and rule 2 checks 2 different cases. Image the nonterminal as an internal node in the parse tree. Rule 1 checks whether this tree can be reached from the root of the tree, while rule 2 checks whether there's a path from this node to the leaf.

Eliminate Useless Nonterms

Grows useful sets.

If a nonterminal cannot derive a sequence of terminal symbols, then it is useless

```
Mark all terminal symbols
Repeat
   If all symbols on the
   righthand side of a
   production are marked
      mark the lefthand side
Until no more non-terminals
can be marked
```

Example:

S	$\longrightarrow$	X | Y
X	$\longrightarrow$	( )
Y	$\longrightarrow$	(Y Y)

S -> X
S -> Y

{(,)} -> {X, (,)} -> {S, X, (,)}.
So the useful non-terminals are S, X;
Y is useless.

Eliminate Useless Nonterms

Intuitiion: Doing a DFS on the grammar from the start symbol.

Keeps growing the set of useful nonterms.

If a nonterminal cannot be derived from the start symbol, then it is useless

```
Mark the start symbol
Repeat
    If the lefthand side of a
    production is marked
        mark all righthand
        non-terminal
Until no more non-terminals
can be marked
```

Example:

S	$\longrightarrow$	A B
A	$\longrightarrow$	**+** | **-** | ε
B	$\longrightarrow$	**digit** | B **digit**
C	$\longrightarrow$	**.** B

{S} -> {S, A, B}

Chomsky Normal Form

4 Steps

Eliminating epsilon rules and eliminating unit rules are very similar.

- Eliminate epsilon rules
- Eliminate unit rules

Only one nonterms on the right side of the production rule.

- Fix productions with terminals on RHS
- Fix productions with > 2 nonterminals on RHS

Eliminate (Most) Epsilon Productions

If a nonterminal *A* immediately derives epsilon

- Make copies of all rules with *A* on the RHS and delete all combinations of *A* in those copies

Example 1

F	→	**id** (A)
A	→	ε
A	→	N
N	→	**id**
N	→	**id ,** N

Like lifting the epsilon symbol up by one level.

$$\Downarrow$$

F	→	**id** (A)
F	→	**id** ( )
A	→	N
N	→	**id**
N	→	**id ,** N

Example 2

$$\begin{aligned}
X &\longrightarrow A\ \mathbf{x}\ A\ \mathbf{y}\ A \\
A &\longrightarrow \varepsilon \\
A &\longrightarrow \mathbf{z}
\end{aligned}$$

$$\begin{aligned}
X &\longrightarrow A\ \mathbf{x}\ A\ \mathbf{y}\ A \\
&\quad |\ \ A\ \mathbf{x}\ A\ \mathbf{y} \\
&\quad |\ \ A\ \mathbf{x}\ \mathbf{y}\ A \\
&\quad |\ \ A\ \mathbf{x}\ \mathbf{y} \\
&\quad |\ \ \mathbf{x}\ A\ \mathbf{y}\ A \\
&\quad |\ \ \mathbf{x}\ A\ \mathbf{y} \\
&\quad |\ \ \mathbf{x}\ \mathbf{y}\ A \\
&\quad |\ \ \mathbf{x}\ \mathbf{y} \\
A &\longrightarrow \mathbf{z}
\end{aligned}$$

The number of new rules is 2 ^ (number of A's). This is exponential. Would this be a problem?

(1) In practical world, grammar is usually short;
(2) Constant time actually, if the grammar is fixed. It would not grow.

Eliminate Unit Productions

Productions of the form $A \longrightarrow B$ are called unit productions

Place B anywhere A could have appeared and remove the unit production

Example 1

F	→	**id (** A **)**
F	→	**id ( )**
A	→	N
N	→	**id**
N	→	**id ,** N

$$\Downarrow$$

F	→	**id (** N **)**
F	→	**id ( )**
N	→	**id**
N	→	**id ,** N

Fix RHS Terminals

RHS has both terminals and non-terminals. Replace each terminal with a non-terminal, to make it all non-terminal.

For productions with Terminals and something else on the RHS

– For each terminal **t** add the rule

$X \longrightarrow \mathbf{t}$

Where X is a new non-terminal

– Replace t with X in the in the original rules

Example

F	→	**id** (N)
F	→	**id** ( )
N	→	**id**
N	→	**id** , N

Renaming.

Not changed to avoid unit rules.

F	→	I L N R
F	→	I L R
N	→	**id**
N	→	I C N
I	→	**id**
L	→	(
R	→	)
C	→	,

Fix RHS Nonterminals

For productions with > 2 Nonterminals on the RHS

- Replace all but the *first* nonterminal with a new nonterminal
- Add a rule from the new nonterminal to the replaced nonterminal sequence
- Repeat

Example

$F \longrightarrow I\ L\ N\ R$

$F \longrightarrow I\ W$

$W \longrightarrow L\ N\ R$

$F \longrightarrow I\ W$

$W \longrightarrow L\ X$

$X \longrightarrow N\ R$

Parsing is Tough

CYK parses an arbitrary CFG, but

- $O(n^3)$
- Too slow!

For special class of grammars

- $O(n)$
- Includes LL(1) and LALR(1)

Classes of Grammars

- LL(1)
  - Scans input from Left-to-right (first L)
  - Builds a Leftmost Derivation (second L)
  - Can peek (1) token ahead of the token being parsed
  - Top-down “predictive parsers”
- LALR(1)
  - Uses special lookahead procedure (LA)
  - Scans input from Left-to-right (second L)
  - Rightmost derivation (R)
  - Can also peek (1) token ahead
- LALR(1) strictly more powerful, much harder to understand

In summary

We talked about how to parse with CYK and Chomsky Normal Form grammars