Introduction to Compiler Design

Lesson 9:

Parsers – Parsing Techniques - CYK

Grammars

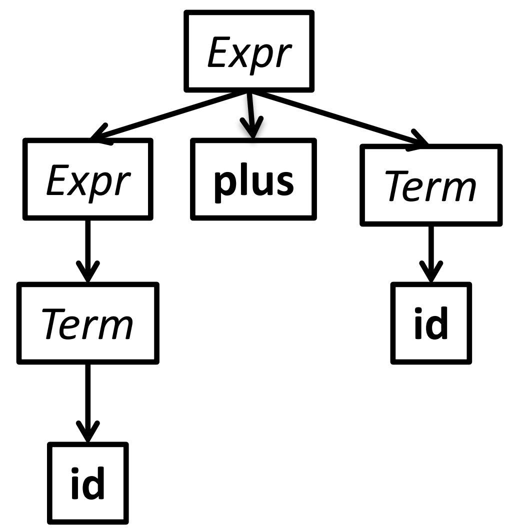
- Context-free grammars (CFGs)
  - Generation: $G \to L(G)$
  - Recognition: Given w, is $w \in L(G)$?
- Translation
  - Given $w \in L(G)$, create a parse tree for w
  - Given $w \in L(G)$, create an AST for w
  - The AST is passed to the next component of our compiler

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, but the algorithm is harder to understand
- Java CUP generates a LALR(1) parser

Approaches to Parsing

- Top Down / “Goal driven”
  - Begin with the start nonterminal
  - Grow parse tree downward to match the string
- Bottom Up / “Data Driven”
  - Start at terminals
  - Generate ever larger subtrees; the goal is to obtain a single tree whose root is the start nonterminal



Parsing Algorithms

- Top-down (“recursive-descent”) for LL(1) grammars
  - How to parse, given the appropriate parse table for G
  - How to construct the parse table for G
- Bottom-up for LALR(1) grammars
  - How to parse, given the appropriate parse table for G
  - How to construct the parse table for G
- CYK

Parser Operations

- Top-down parser
  - *Scan* the next input token
  - *Pop* a single symbol
  - *Push* a bunch of RHS symbols
- Bottom-up parser
  - *Shift* an input token into a stack item
  - *Reduce* a bunch of stack items into a new parent item (and push the parent on the stack)

Top-Down Parsers

- Start at the **Start** symbol
- Repeatedly: “predict” what production to use
  - Example: if the current token to be parsed is an **id**, no need to try productions that start with **intLiteral**
  - This might seem simple, but keep in mind that a chain of productions may have to be used to get to the rule that handles, e.g., **id**

Restricting the Grammar

- By restricting our grammars we can
  - Detect ambiguity
  - Build linear-time, O(n) parsers
- LL(1) languages
  - Particularly amenable to parsing
  - Parsable by <u>predictive</u> (top-down) parsers (sometimes called “recursive-descent parsers”)

LL(1) Grammar Transformations

- Necessary (but not sufficient conditions) for LL(1) parsing:
  - Free of left recursion
    - No left-recursive rules
    - Why? Need to look past the list to know when to cap it
  - Left-factored
    - No rules with a common prefix, for any nonterminal
    - Why? We would need to look past the prefix to pick the production

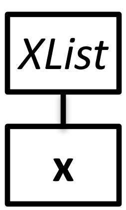
Why Left Recursion is a Problem

CFG snippet: $XList \longrightarrow XList\ \mathbf{x}\ |\ \mathbf{x}$

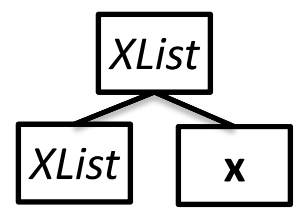
Current parse tree: $\boxed{XList}$

Current token: **x**

How should we grow the tree top-down?



(OR)



Correct if there are no more **x**s

Correct if there <u>are</u> more **x**s

We don't know which to choose without more lookahead

Left-Recursion Elimination: Review

Replace $A \longrightarrow A\ \alpha \mid \beta$

Head of the list

With $A \longrightarrow \beta\ A'$

$A' \longrightarrow \alpha\ A' \mid \varepsilon$

Where β does not start with *A, or* may not be present

Preserves the language (a list of αs, starting with a β), but uses right recursion

Left-Recursion Elimination: Ex1

$$A \longrightarrow A\,\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \longrightarrow \beta\, A' \\ A' \longrightarrow \alpha\, A' \mid \varepsilon \end{array}$$

$$E \longrightarrow E\, \underbrace{\textbf{cross id}}_{\alpha} \mid \underbrace{\textbf{id}}_{\beta} \quad \Rightarrow \quad \begin{array}{l} E \longrightarrow \overbrace{\textbf{id}}^{\beta}\, E' \\ E' \longrightarrow \underbrace{\textbf{cross id}}_{\alpha}\, \mathrm{E}' \mid \varepsilon \end{array}$$

Left-Recursion Elimination: Ex2

$$A \longrightarrow A\,\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \longrightarrow \beta\, A' \\ A' \longrightarrow \alpha\, A' \mid \varepsilon \end{array}$$

$$\begin{array}{l} E \longrightarrow E \,\mathbf{+}\, T \mid T \\ T \longrightarrow T \,\mathbf{*}\, F \mid F \\ F \longrightarrow \mathbf{(}\ \mathrm{E}\ \mathbf{)} \mid \mathbf{id} \end{array} \quad \Rightarrow \quad \begin{array}{l} E \longrightarrow T\, E' \\ E' \longrightarrow \mathbf{+}\ \mathrm{T\ E'} \mid \varepsilon \\ T \longrightarrow F\, T' \\ T' \longrightarrow \mathbf{*}\ \mathrm{F\ T'} \mid \varepsilon \\ F \longrightarrow \mathbf{(}\ \mathrm{E}\ \mathbf{)} \mid \mathbf{id} \end{array}$$

Left-Recursion Elimination: Ex3

$A \longrightarrow A\ \alpha \mid \beta$ ⇒ $A \longrightarrow \beta\ A'$
$A' \longrightarrow \alpha\ A' \mid \varepsilon$

DList ⟶ *DList D* | ε
D ⟶ *Type* **id semi**
Type ⟶ **bool** | **int**

⇒

DList ⟶ ε *DList'*
DList' ⟶ *D DList'* | ε
D ⟶ *Type* **id semi**
Type ⟶ **bool** | **int**

⇒

DList ⟶ *D DList* | ε
D ⟶ *Type* **id semi**
Type ⟶ **bool** | **int**

Left Factoring

Removing a common prefix from a grammar

Replace $A \longrightarrow \boxed{\alpha\, \beta_1 \mid \ldots \mid \alpha\, \beta_m} \mid y_1 \mid \ldots \mid y_n$

With $A \longrightarrow \boxed{\alpha\, A'} \mid y_1 \mid \ldots \mid y_n$

$A' \longrightarrow \beta_1 \mid \ldots \mid \beta_m$

Where β_i and y_i are sequence of symbols with no common prefix
Note: y_i may not be present, and one of the β may be ε

Combine all "problematic" rules that start with α into one rule $\alpha\, A'$
Now A' represents the suffix of the "problematic" rules

Left Factoring: Example 1

$$A \longrightarrow \alpha\,\beta_1 \mid \dots \mid \alpha\,\beta_m \mid \gamma_1 \mid \dots \mid \gamma_n \quad \Rightarrow \quad \begin{array}{l} A \longrightarrow \alpha\,A' \mid \gamma_1 \mid \dots \mid \gamma_n \\ A' \longrightarrow \beta_1 \mid \dots \mid \beta_m \end{array}$$

$$X \longrightarrow \overbrace{\mathbf{<}}^{\alpha}\,\overbrace{\mathbf{a}\ \mathbf{>}}^{\beta_1} \mid \overbrace{\mathbf{<}}^{\alpha}\,\overbrace{\mathbf{b}\ \mathbf{>}}^{\beta_2} \mid \overbrace{\mathbf{<}}^{\alpha}\,\overbrace{\mathbf{c}\ \mathbf{>}}^{\beta_3} \mid \overbrace{\mathbf{d}}^{\gamma_1}$$

$$X \longrightarrow \overbrace{\mathbf{<}}^{\alpha}\,X' \mid \overbrace{\mathbf{d}}^{\gamma_1}$$

$$X' \longrightarrow \underbrace{\mathbf{a}\ \mathbf{>}}_{\beta_1} \mid \underbrace{\mathbf{b}\ \mathbf{>}}_{\beta_2} \mid \underbrace{\mathbf{c}\ \mathbf{>}}_{\beta_3}$$

Left Factoring: Example 2

$$A \longrightarrow \alpha\,\beta_1 \mid \dots \mid \alpha\,\beta_m \mid \gamma_1 \mid \dots \mid \gamma_n \quad \Rightarrow \quad \begin{array}{l} A \longrightarrow \alpha\,A' \mid \gamma_1 \mid \dots \mid \gamma_n \\ A' \longrightarrow \beta_1 \mid \dots \mid \beta_m \end{array}$$

$$Stmt \longrightarrow \textbf{id}\ \overbrace{\textbf{assign}\ E}^{\beta_1} \mid \textbf{id}\ \overbrace{\textbf{(}\ EList\ \textbf{)}}^{\beta_2} \mid \textbf{return}$$

$$E \longrightarrow \textbf{intlit} \mid \textbf{id}$$

$$Elist \longrightarrow E \mid E\ \textbf{comma}\ EList$$

$$\text{Stmt} \longrightarrow \textbf{id}\ Stmt' \mid \textbf{return}$$

$$\text{Stmt}' \longrightarrow \textbf{assign}\ E \mid \textbf{(}\ EList\ \textbf{)}$$

$$E \longrightarrow \textbf{intlit} \mid \textbf{id}$$

$$Elist \longrightarrow E \mid E\ \textbf{comma}\ EList$$

Left Factoring: Example 3

$$A \longrightarrow \alpha\,\beta_1 \mid \dots \mid \alpha\,\beta_m \mid y_1 \mid \dots \mid y_n \quad \Rightarrow \quad \begin{array}{l} A \longrightarrow \alpha\,A' \mid y_1 \mid \dots \mid y_n \\ A' \longrightarrow \beta_1 \mid \dots \mid \beta_m \end{array}$$

$$S \longrightarrow \overbrace{\textbf{if}\ E\ \textbf{then}\ S}^{\alpha}\,\overbrace{\vphantom{S}}^{\beta_1 = \varepsilon} \mid \overbrace{\textbf{if}\ \mathrm{E}\ \textbf{then}\ S}^{\alpha}\ \overbrace{\textbf{else}\ S}^{\beta_2} \mid \textbf{semi}$$

$$E \longrightarrow \textbf{boollit}$$

$$\mathrm{S} \longrightarrow \textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{S}\ \mathrm{S'} \mid \textbf{semi}$$

$$\mathrm{S'} \longrightarrow \textbf{else}\ \mathrm{S} \mid \varepsilon$$

$$\mathrm{E} \longrightarrow \textbf{boollit}$$

Left Factoring: Not Always Immediate

$$A \longrightarrow \alpha\,\beta_1 \mid \ldots \mid \alpha\,\beta_m \mid y_1 \mid \ldots \mid y_n$$

$$\Rightarrow$$

$$A \longrightarrow \alpha\, A' \mid y_1 \mid \ldots \mid y_n$$
$$A' \longrightarrow \beta_1 \mid \ldots \mid \beta_m$$

This snippet yearns for left factoring

S ⟶ A | C | **return**
A ⟶ **id assign** *E*
C ⟶ **id (** *EList* **)**

but we cannot! At least without *inlining*

S ⟶ **id assign** *E* | **id (** *Elist* **)** | **return**

Some Interesting Properties of CYK

- Very old algorithm
  - Already well known in early 70s
- No problems with ambiguous grammars:
  - Gives a solution for *all* possible parse tree simultaneously

LL(1) Not Powerful Enough for all PL

- Left-recursion
- Not left factored
- Doesn't mean LL(1) is bad
  - Right tool for simple parsing jobs

```
stmtList  ::= stmtList stmt
           |  /* epsilon */
           ;
```

We Need a *Little* More Power

- Could increase the lookahead
  - Up until the mid 90s, this was considered impractical
- Could increase the runtime complexity
  - CYK
- Could increase the memory complexity
  - i.e., create a more elaborate parse table

LR Parsers

- Left-to-right scan of the input file
- Reverse-rightmost derivation
- Advantages
  - Can recognize almost any programming language
  - Time and space O(n) in the input size
  - LR parsers more powerful than LL parser: LL(1) $\subset$ LR(1)
- Disadvantages
  - More complex parser generation
  - Larger parse tables

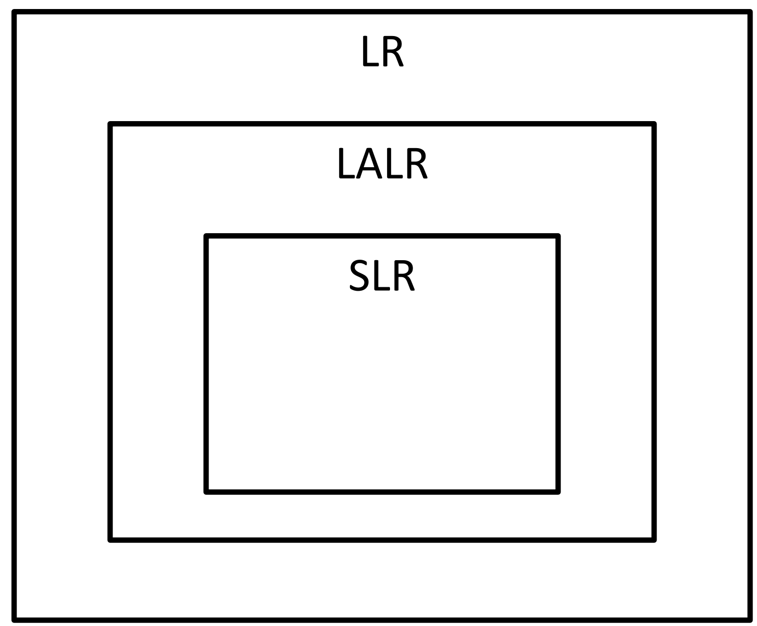
LR Parser Power

- Let $S \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow w$ be a rightmost derivation, where ω is a terminal string
- Let $\alpha A\gamma \Longrightarrow \alpha\beta\gamma$ be a step in the derivation
    - So $A \longrightarrow \beta$ must have been a production in the grammar
    - $\alpha\beta\gamma$ must be some α_i or w
  - A grammar is LR(k) if for every derivation step, $A \longrightarrow \beta$ can be inferred using only a scan of αβ and at most k symbols of γ
- Much like LL(1), you generally just have to go ahead and try it

LR Parser types

Recognizable by a deterministic PDA

- LR(1)
  - Can recognize any DCFG
  - Can experience blowup in parse table size
- LALR(1)
- SLR(1)
  - Both proposed at the same time to limit parse table size



How Does Bottom-Up Parsing Work?

- One example follows: CYK
  - Simultaneously tracked every possible parse tree
  - LR parsers work in a similar way
- Contrast to top-down parser
  - We know exactly where we are in the parse
  - Make predictions about what’s next

CYK: A General Approach to Parsing

- Cocke–Younger–Kasami algorithm
- Operates in time $O(n^3)$
- Works bottom-up
- Requires the grammar to be in Chomsky Normal Form
  - This turns out not to be a limitation: any context-free grammar can be converted into one in Chomsky Normal Form

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

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

s = s1 s2 s3 s4

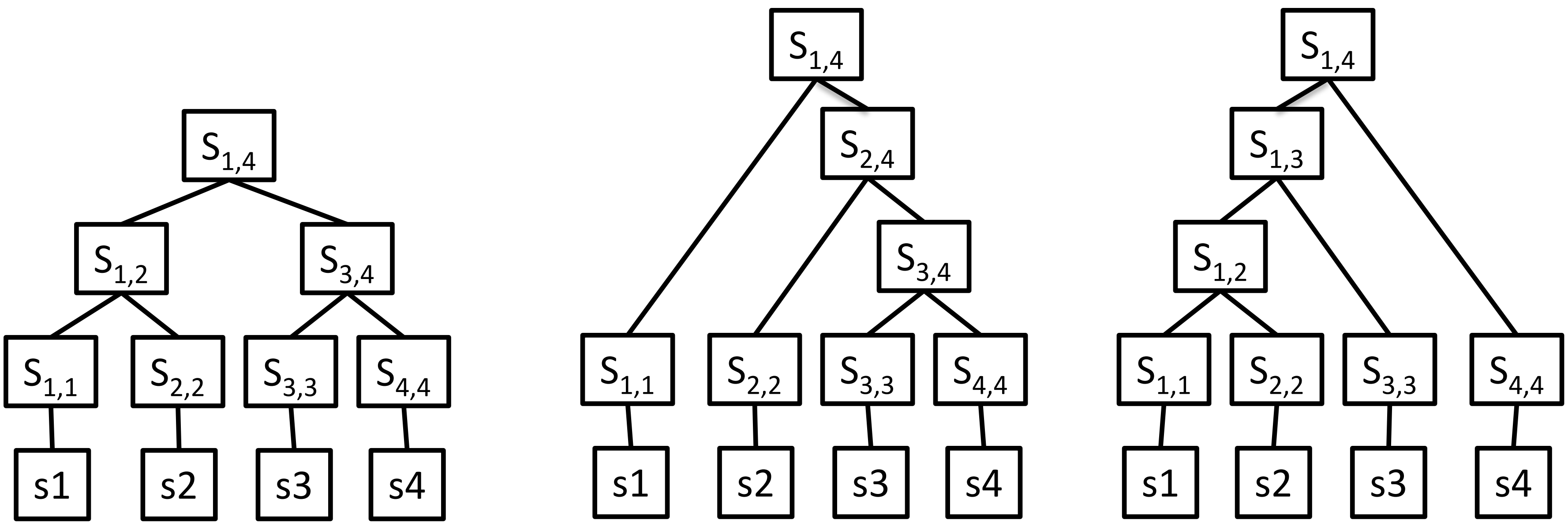
CYK: Dynamic Programming

$X \longrightarrow \mathbf{t}$

Form the leaves of the parse tree

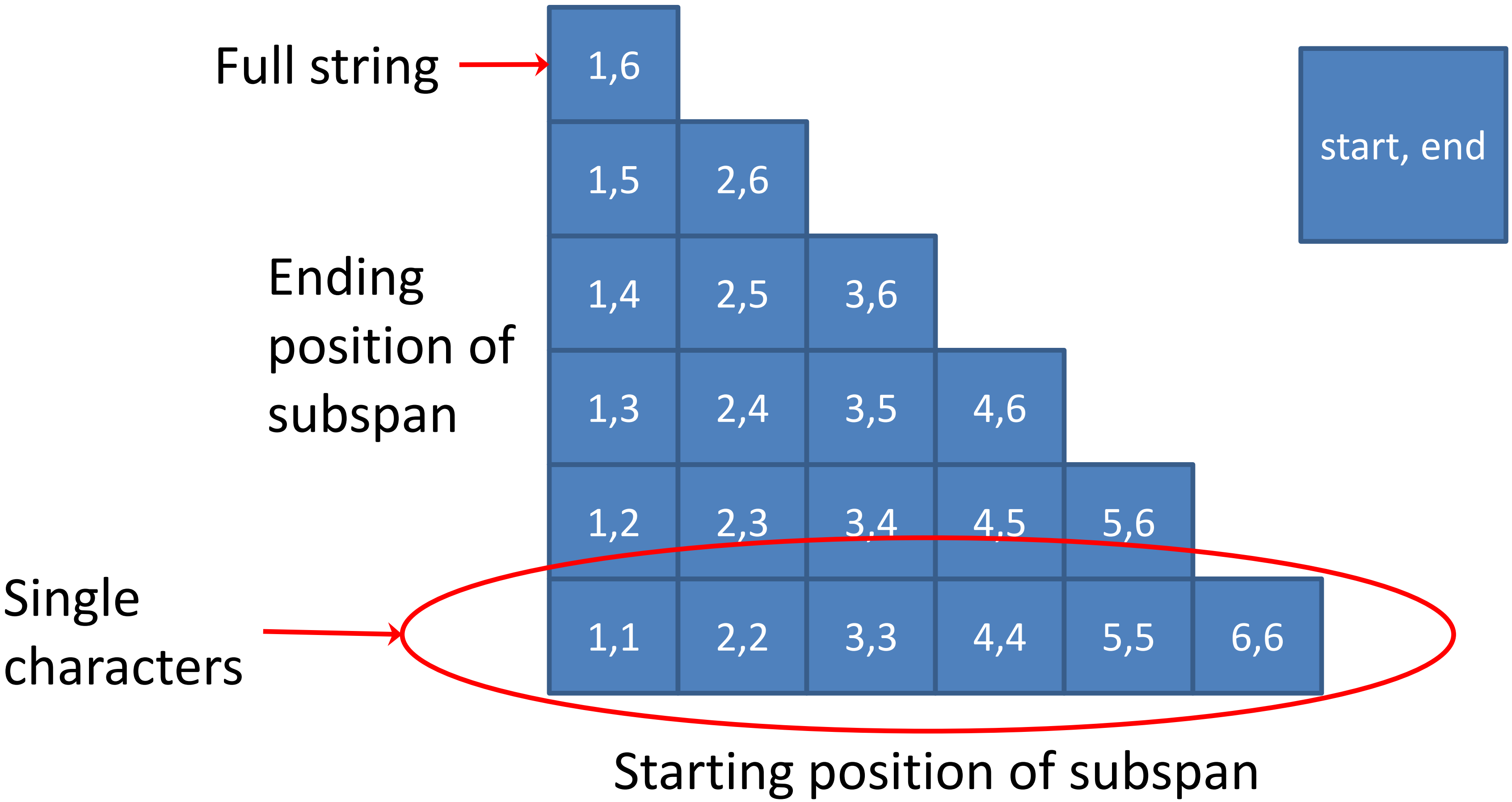
$X \longrightarrow A\ B$

Form binary interior nodes of the parse tree

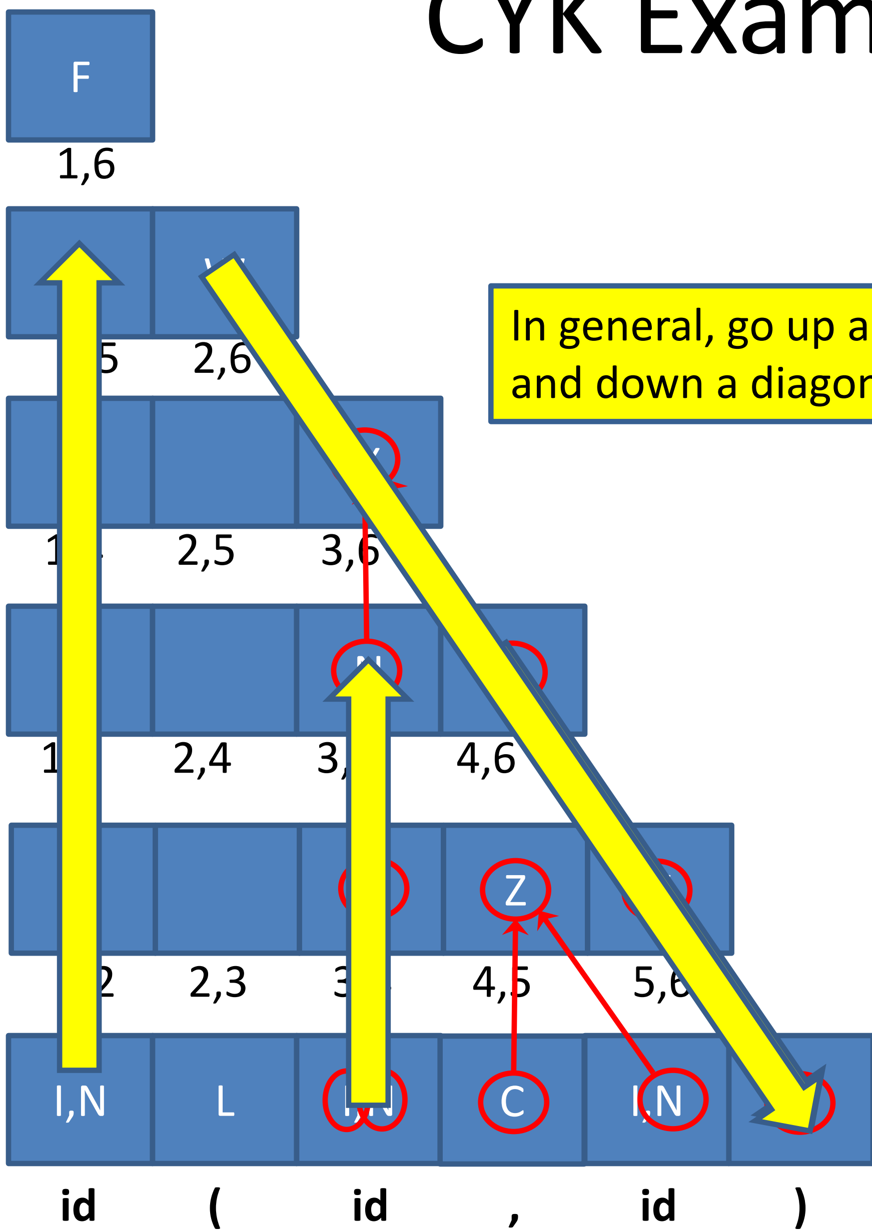


Running CYK

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



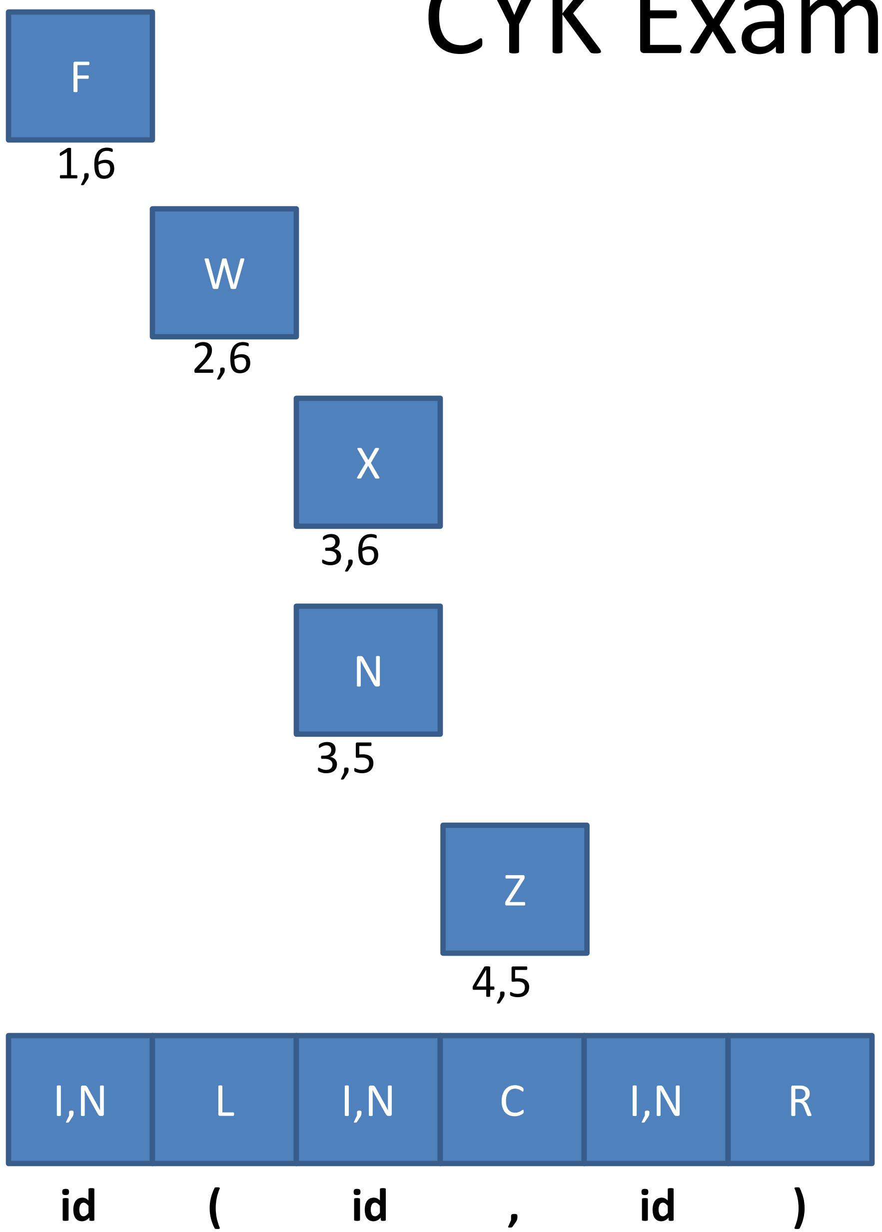
CYK Example



In general, go up a column and down a diagonal

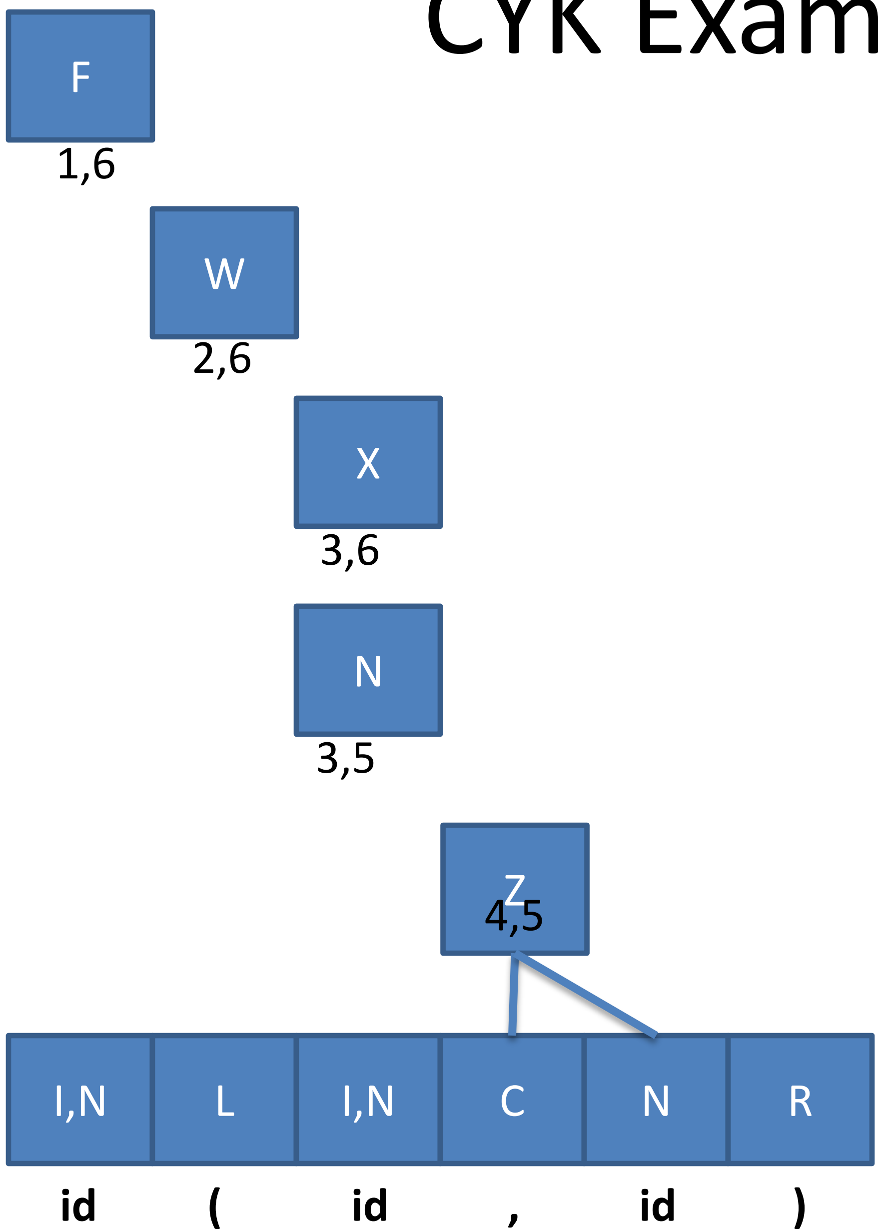
F	$\longrightarrow$	I W
F	$\longrightarrow$	I Y
W	$\longrightarrow$	L X
X	$\longrightarrow$	N R
Y	$\longrightarrow$	L R
N	$\longrightarrow$	**id**
N	$\longrightarrow$	I Z
Z	$\longrightarrow$	C N
I	$\longrightarrow$	**id**
L	$\longrightarrow$	**(**
R	$\longrightarrow$	**)**
C	$\longrightarrow$	**,**

CYK Example



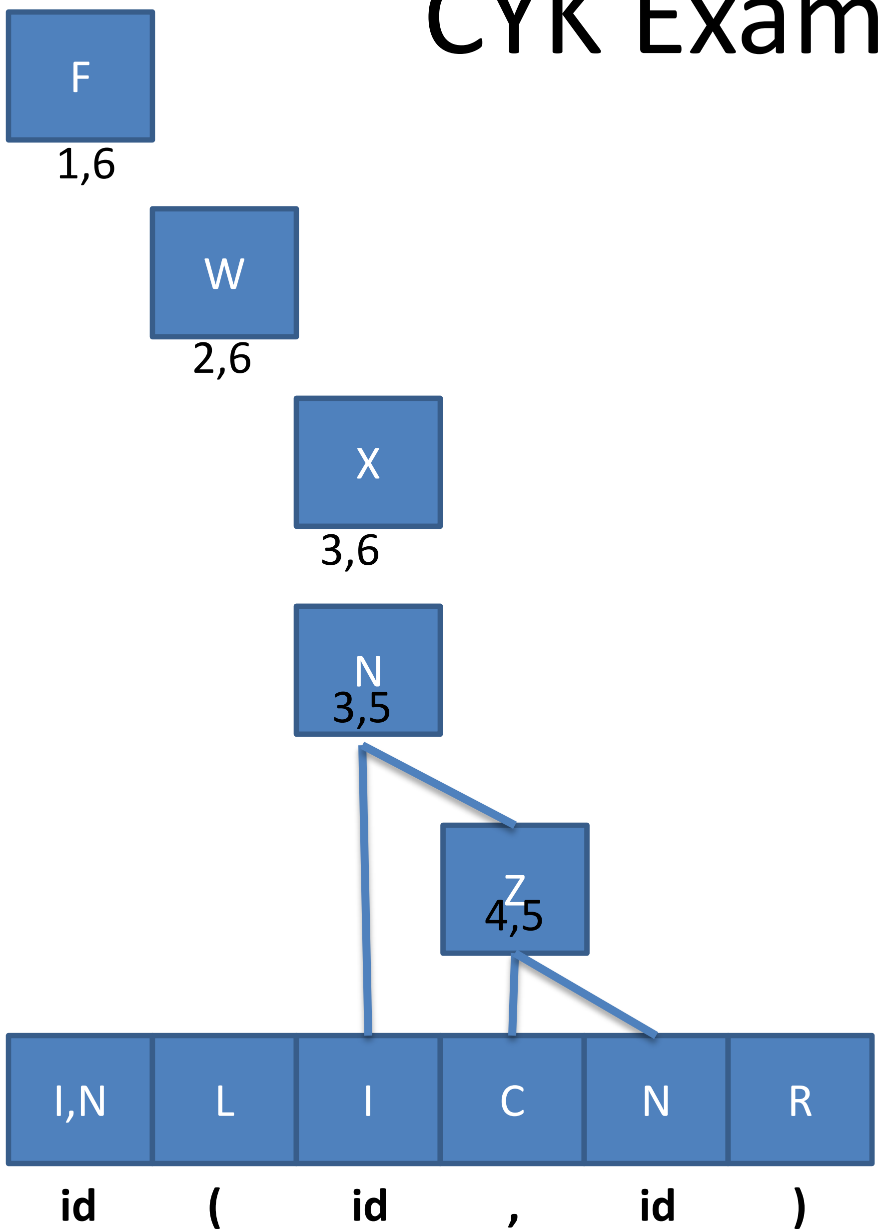
F	⟶	I W
F	⟶	I Y
W	⟶	L X
X	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	**(**
R	⟶	**)**
C	⟶	**,**

CYK Example



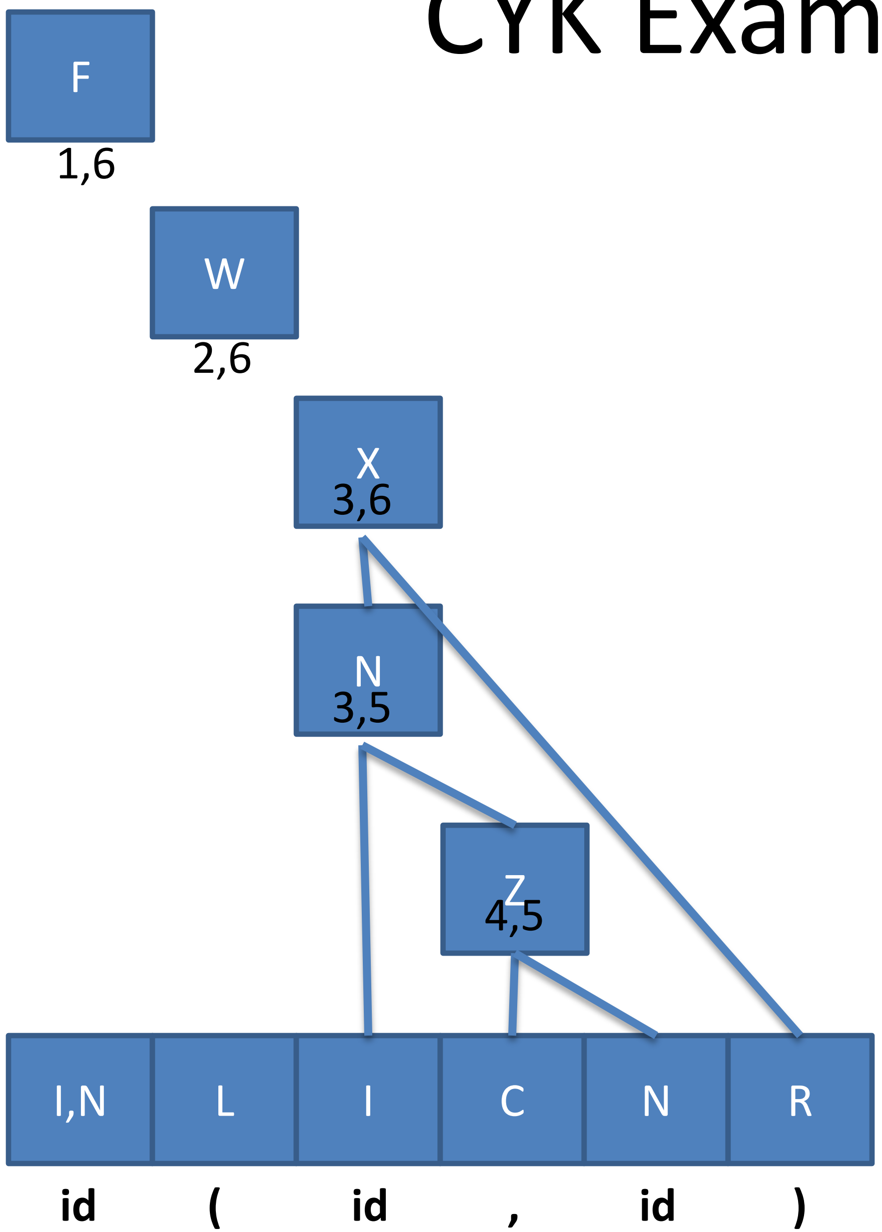
F	⟶	I W
F	⟶	I Y
W	⟶	L X
X	⟶	N R
Y	⟶	L R
N	⟶	**id**
N	⟶	I Z
Z	⟶	C N
I	⟶	**id**
L	⟶	**(**
R	⟶	**)**
C	⟶	**,**

CYK Example



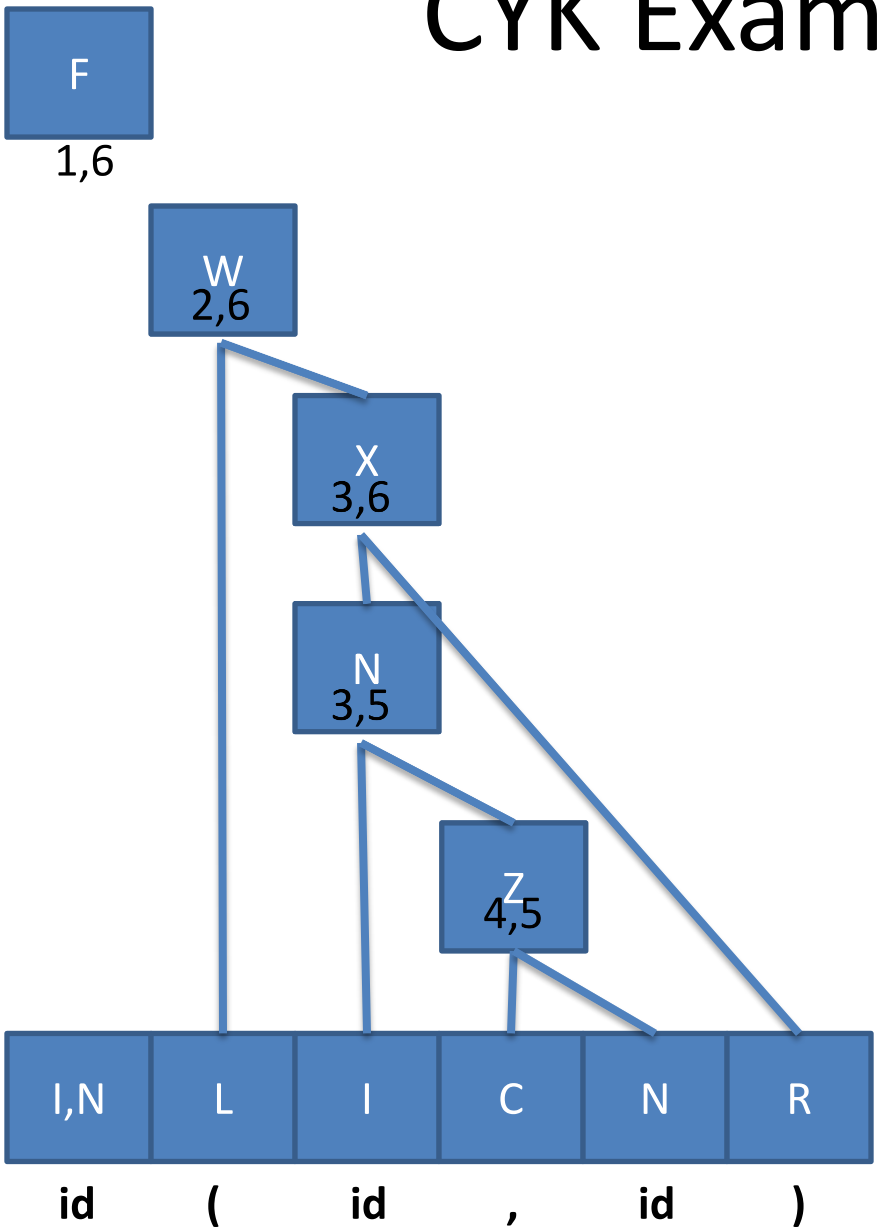
F	$\longrightarrow$	I W
F	$\longrightarrow$	I Y
W	$\longrightarrow$	L X
X	$\longrightarrow$	N R
Y	$\longrightarrow$	L R
N	$\longrightarrow$	**id**
N	$\longrightarrow$	I Z
Z	$\longrightarrow$	C N
I	$\longrightarrow$	**id**
L	$\longrightarrow$	**(**
R	$\longrightarrow$	**)**
C	$\longrightarrow$	**,**

CYK Example



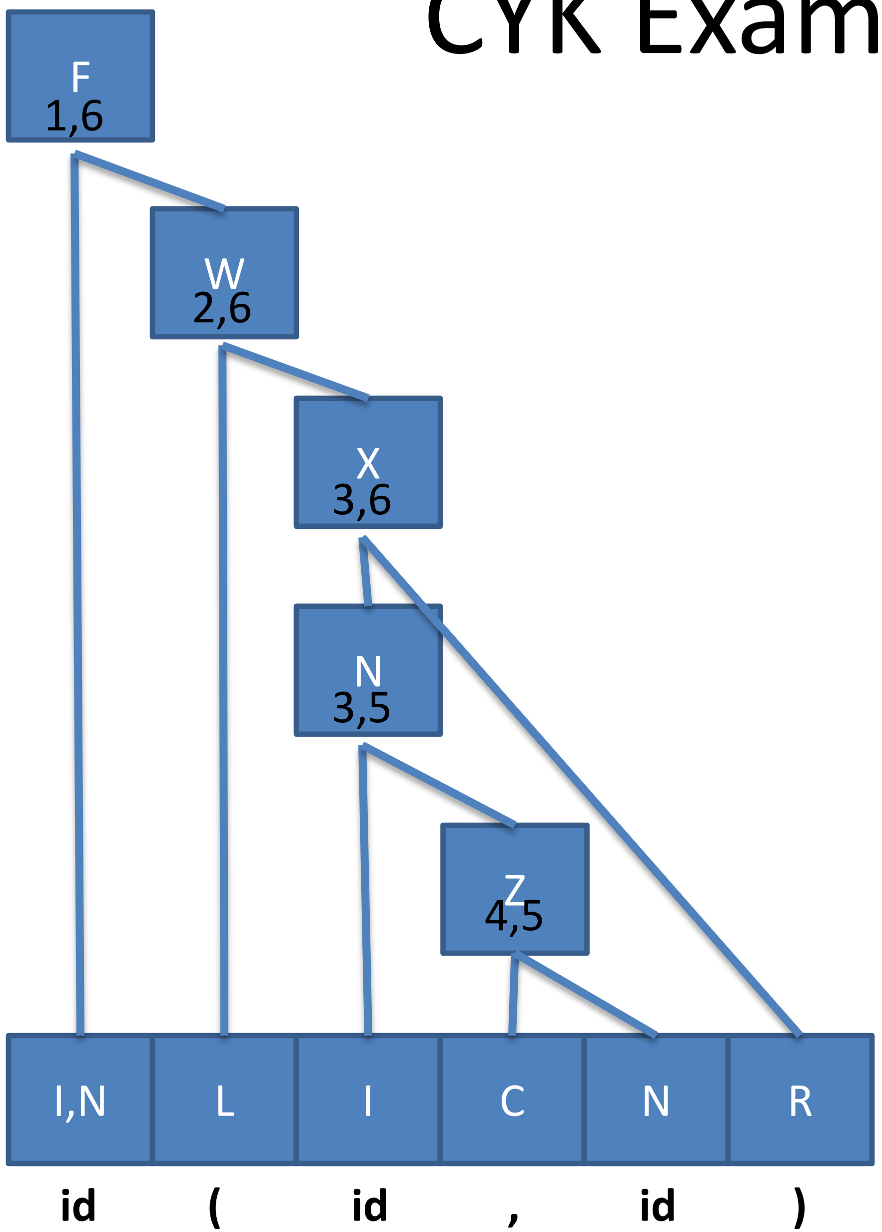
F $\longrightarrow$ I W
F $\longrightarrow$ I Y
W $\longrightarrow$ L X
X $\longrightarrow$ N R
Y $\longrightarrow$ L R
N $\longrightarrow$ **id**
N $\longrightarrow$ I Z
Z $\longrightarrow$ C N
I $\longrightarrow$ **id**
L $\longrightarrow$ **(**
R $\longrightarrow$ **)**
C $\longrightarrow$ **,**

CYK Example



F	$\longrightarrow$	I W
F	$\longrightarrow$	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



F	⟶	I W
F	⟶	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*

Eliminate Useless Nonterminals

- 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:

$$S \longrightarrow X \mid Y$$

$$X \longrightarrow (\ )$$

$$Y \longrightarrow (\ Y\ Y\ )$$

Eliminate Useless Nonterminals

- 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

Chomsky Normal Form

- 4 Steps
  - Eliminate epsilon rules
  - Eliminate unit rules
  - 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

⇓

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

Example 2

$$X \longrightarrow A\ \mathbf{x}\ A\ \mathbf{y}\ A$$
$$A \longrightarrow \varepsilon$$
$$A \longrightarrow \mathbf{z}$$

$$\begin{array}{lcl}
X & \longrightarrow & A\ \mathbf{x}\ A\ \mathbf{y}\ A \\
 & | & A\ \mathbf{x}\ A\ \mathbf{y} \\
 & | & A\ \mathbf{x}\ \mathbf{y}\ A \\
 & | & \mathbf{x}\ A\ \mathbf{y}\ A \\
 & | & A\ \mathbf{x}\ \mathbf{y} \\
 & | & \mathbf{x}\ A\ \mathbf{y} \\
 & | & \mathbf{x}\ \mathbf{y}\ A \\
 & | & \mathbf{x}\ \mathbf{y} \\
A & \longrightarrow & \mathbf{z}
\end{array}$$

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

F	⟶	**id (** N **)**
F	⟶	**id ( )**
N	⟶	**id**
N	⟶	**id ,** N

Fix RHS Terminals

- 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 original rules

Example

F	$\longrightarrow$	**id (** N **)**
F	$\longrightarrow$	**id ( )**
N	$\longrightarrow$	**id**
N	$\longrightarrow$	**id ,** N

F	$\longrightarrow$	I L N R
F	$\longrightarrow$	I L R
N	$\longrightarrow$	**id**
N	$\longrightarrow$	I C N
I	$\longrightarrow$	**id**
L	$\longrightarrow$	**(**
R	$\longrightarrow$	**)**
C	$\longrightarrow$	**,**

Fix RHS non-terminals

- For productions with more than two non-terminals 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

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

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

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

Some Final Thoughts on LR Parsing

- A bit complicated to build the parse table
  – Fortunately, algorithms exist
- Still not as powerful as CYK
  – Shift/reduce: action table cell includes S and R
  – Reduce/reduce: cell include > 1 R rule
- SDT similar to LL(1)
  – Embed SDT action numbers in action table
  – Fire off on reduce rules