Appendix: Syntax Analysis

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Stephen Chong's slides (CS153 Compilers) and Modern Compiler Implementation in C (Andrew W. Appel)

Parsing

- Parsing
  - Context-free grammars
  - Derivations
  - Parse trees
  - Ambiguous grammars
  - Recursive descent parsing
  - Parser combinators

Parsing

- Two pieces conceptually:
  - Recognizing syntactically valid phrases.
  - Extracting semantic content from the syntax.
    - E.g., What is the subject of the sentence in English?
    - E.g., Is the syntax ambiguous?
      - ✓ If so, which meaning do we take?
        - “Time flies like an arrow”, “Fruit flies like a banana”
        - “2 * 3 + 4”

- In practice, solve both problems at the same time.

Specifying the Language

- A language is a set of strings. We need to specify what this set is.
- Can we use regular expressions?
  - It’s impossible for finite automaton to recognize language with balanced parentheses!

Context-Free Grammars

- Context Free Grammars (CFGs) are regular expressions with recursion
- CFGs provide declarative specification of syntactic structure
- CFG has set of **productions** of the form
  - symbol → symbol symbol ... symbol

  with zero or more symbols on the right
- Each symbol is either **terminal** (i.e., token from the alphabet) or **non-terminal** (i.e., appears on the LHS of some production)
  - No terminal symbols appear on the LHS of productions

CFG example

$S \rightarrow S;\ S$
$S \rightarrow id := E$
$S \rightarrow print\ (\ L\ )$

$E \rightarrow id$
$E \rightarrow num$
$E \rightarrow E + E$
$E \rightarrow (S, E)$

$L \rightarrow E$
$L \rightarrow L, E$

- Terminals are: **`id print num , + ( ) := ;`**
- Non-terminals are: ***S, E, L***
  - S is the start symbol
- E.g., one sentence in the language is

  `id := num; id := (id := num + num, id+id)`

  - Source text (before lexical analysis) might have been

    `a := 7; b := (c := 30+5, a+c)`

Derivations

- To show that a sentence is in the language of a grammar, we can perform a **derivation**
  - Start with start symbol, repeatedly replace a non-terminal by its right-hand side
- E.g.,

S
S;S
id := E;S
id := E;id := E
id := num; id := E
...
id := num; id := (id := num+num, id+id)

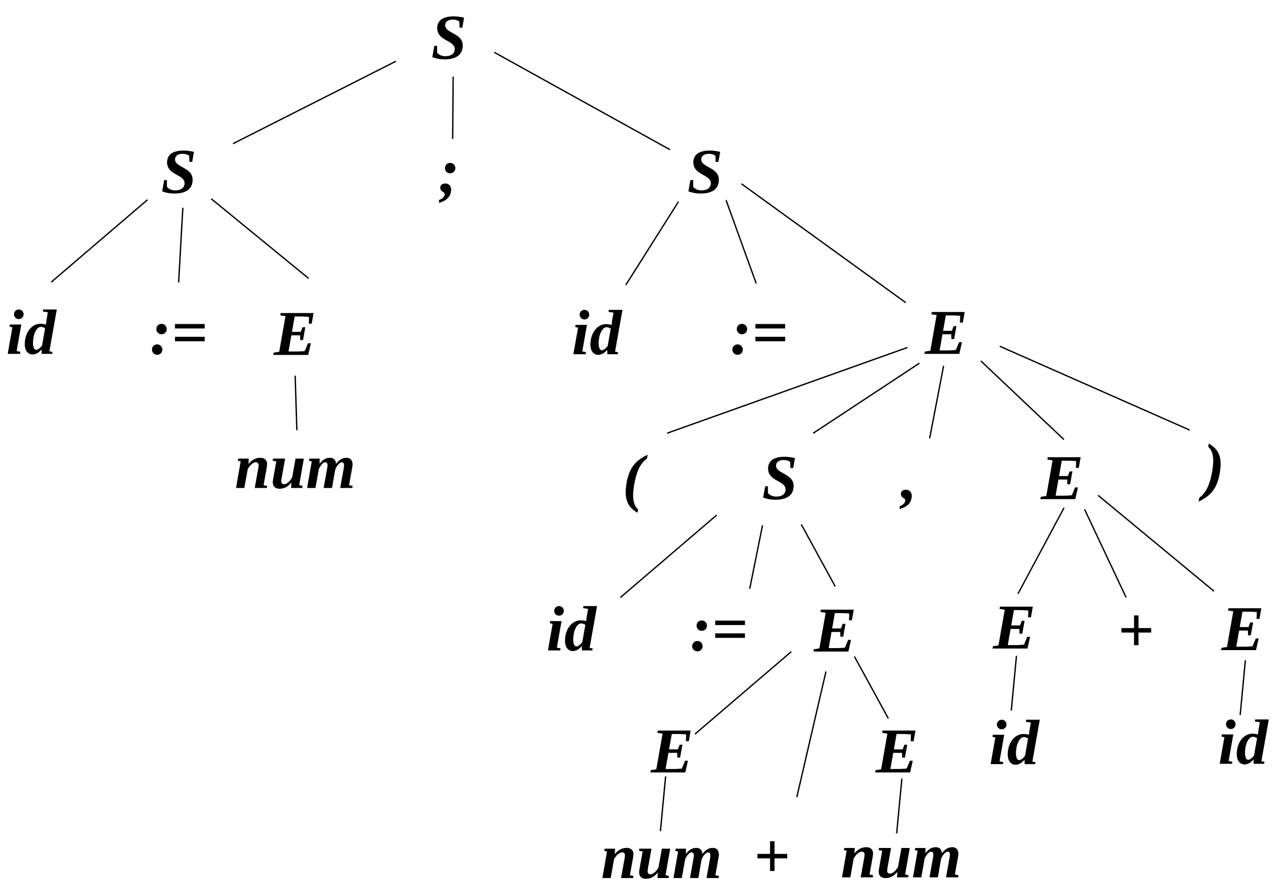
$S \rightarrow S; S$
$S \rightarrow id := E$
$S \rightarrow print (L)$
$E \rightarrow id$
$E \rightarrow num$
$E \rightarrow E + E$
$E \rightarrow (S, E)$
$L \rightarrow E$
$L \rightarrow L, E$

CFGs and Regular Expressions

- CFGs are strictly more expressive than regular expressions

Parse Tree

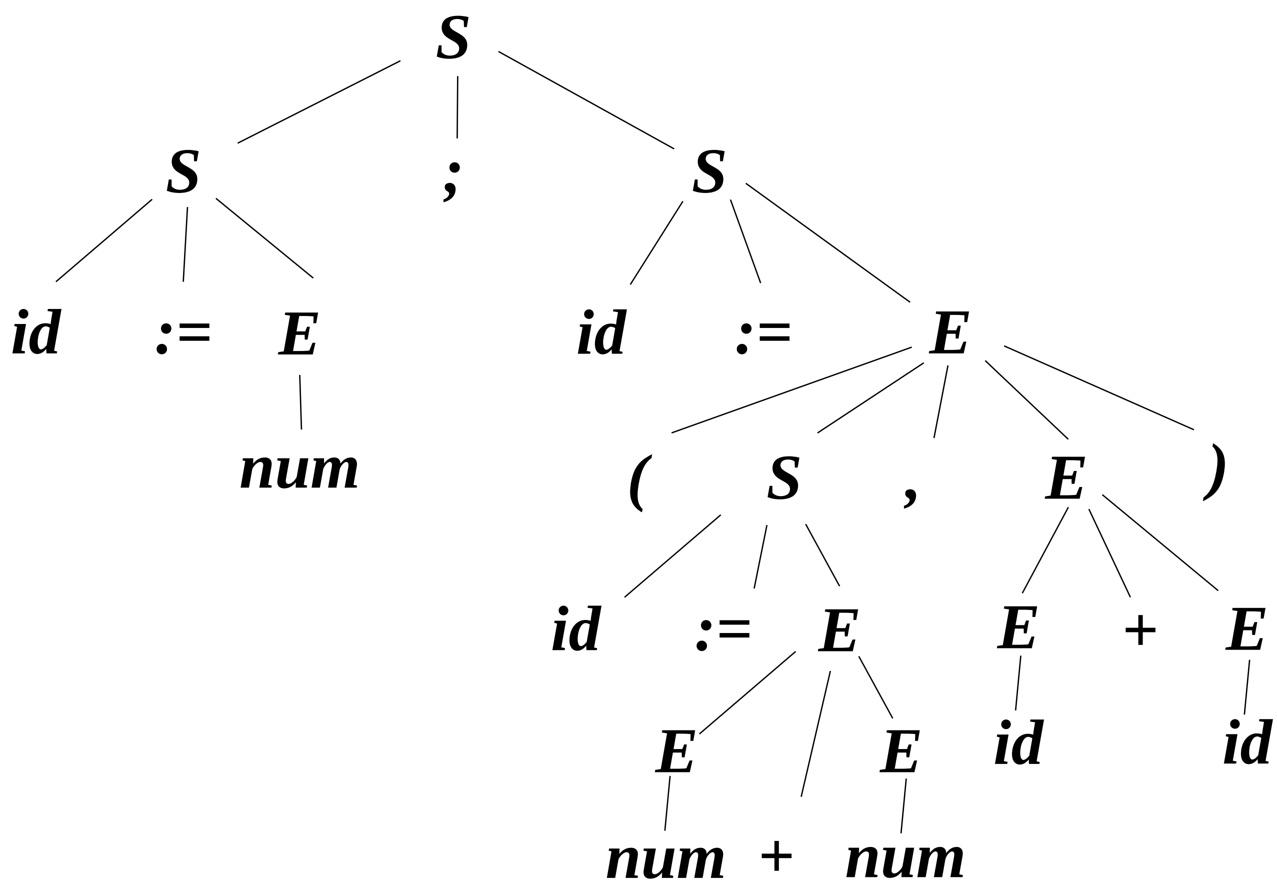
- A **parse tree** connects each symbol to the symbol it was derived from
- A derivation is, in essence, a way of constructing a parse tree.
  - Two different derivations may have the same parse tree



$$
\begin{aligned}
S &\rightarrow S; S \\
S &\rightarrow id := E \\
S &\rightarrow print\ (\ L\ ) \\
E &\rightarrow id \\
E &\rightarrow num \\
E &\rightarrow E + E \\
E &\rightarrow (S, E) \\
L &\rightarrow E \\
L &\rightarrow L, E
\end{aligned}
$$

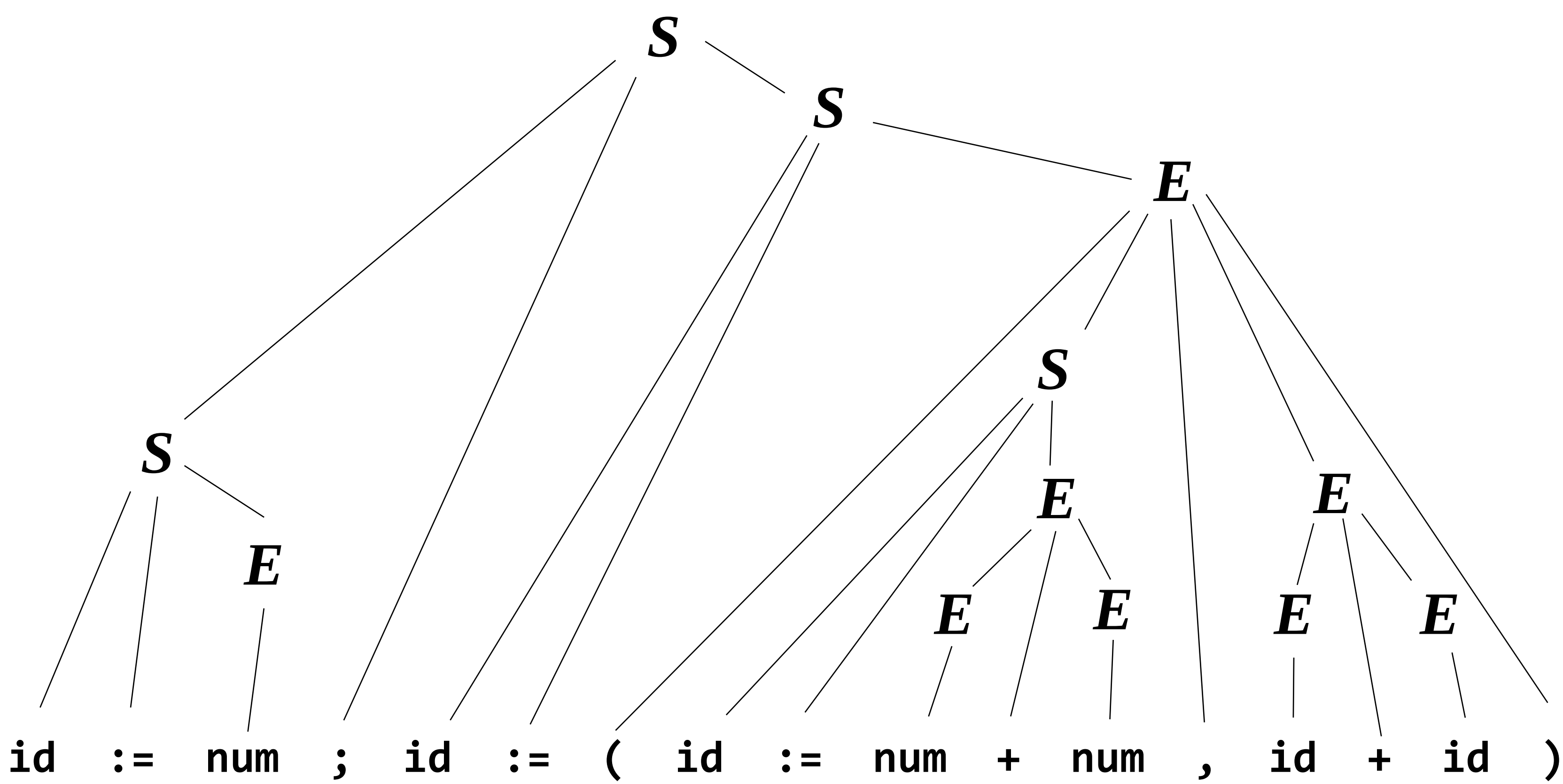
How to Build a Parse Tree / Find a Derivation

- Conceptually, two possible ways:
  - Start from start symbol, choose a non-terminal and expand until you reach the sentence
  - Start from the terminals and replace phrases with non-terminals



How to Build a Parse Tree / Find a Derivation

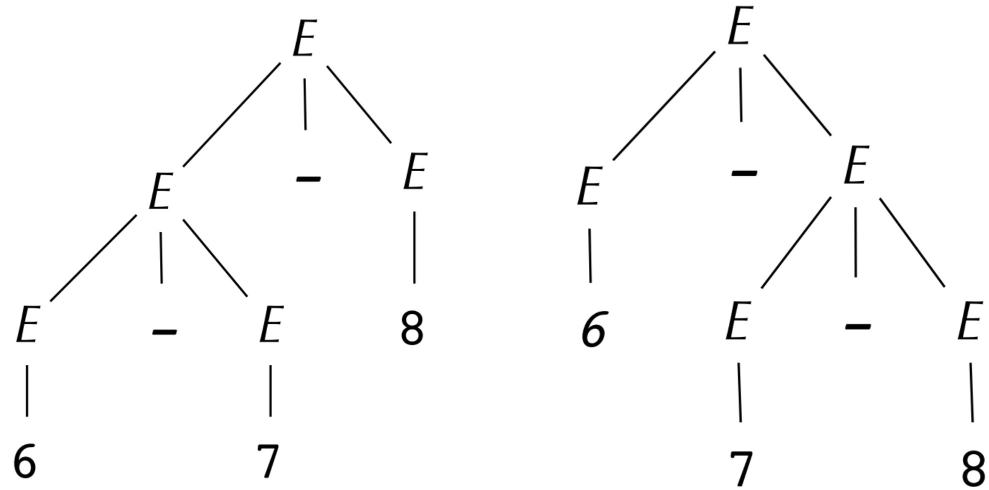
- Conceptually, two possible ways:
  - Start from start symbol, choose a non-terminal and expand until you reach the sentence
  - Start from the terminals and replace phrases with non-terminals



Ambiguous Grammar

- A grammar is **ambiguous** if it can derive a sentence with two different parse trees
- E.g.,

$$E \rightarrow id$$
$$E \rightarrow num$$
$$E \rightarrow E * E$$
$$E \rightarrow E / E$$
$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow (E)$$



- Ambiguity is usual bad: different parse trees often have different meaning!
- But we can usually eliminate ambiguity by transforming the grammar

Fixing Ambiguity Example

- We would like * to **bind higher** than +
  (aka, * to have **higher precedence** than +)
  - So 1+2*3 means 1+(2*3) instead of (1+2)*3
- We would like each operator to **associate to the left**
  - So 6-7-8 means (6-7)-8 instead of 6-(7-8)
- Symbol E for expression, T for term, F for factor

$$E \to E + T \qquad T \to T * F \qquad F \to id$$
$$E \to E - T \qquad T \to T / F \qquad F \to num$$
$$E \to T \qquad T \to F \qquad F \to (E)$$

Left Recursion

- Recursive descent parsing doesn't handle left recursion well!
- We can refactor grammar to avoid left recursion
  - E.g., transform left recursive grammar

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \texttt{id}$$
$$E \rightarrow E - T \qquad T \rightarrow T\ /\ F \qquad F \rightarrow \texttt{num}$$
$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (E)$$

  - to

$$E \rightarrow T\,E' \qquad T \rightarrow F\,T' \qquad F \rightarrow \texttt{id}$$
$$E' \rightarrow +\,T\,E' \qquad T' \rightarrow *\,F\,T' \qquad F \rightarrow \texttt{num}$$
$$E' \rightarrow -\,T\,E' \qquad T' \rightarrow /\,F\,T' \qquad F \rightarrow (E)$$
$$E' \rightarrow \qquad T' \rightarrow$$

Parser Combinators

- Parser combinators are an elegant functional-programming technique for parsing
  - Higher-order functions that accept parsers as input and returns a new parser as output

LL Parsing

- LL Parsing
  - Nullable, First, Follow sets
  - Constructing an LL parsing table

LL(k) Parsing

- Could we somehow know which production to use?
- Basic idea: look at the next k symbols to predict whether we want p_1 or p_2
- How do we predict which production to use?

FIRST Sets

- Given string γ of terminal and non-terminal symbols FIRST(γ) is set of all terminal symbols that can start a string derived from γ

$$
\begin{array}{lll}
E \rightarrow T\, E' & T \rightarrow F\, T' & F \rightarrow \texttt{id} \\
E' \rightarrow +\, T\, E' & T' \rightarrow *\, F\, T' & F \rightarrow \texttt{num} \\
E' \rightarrow -\, T\, E' & T' \rightarrow /\, F\, T' & F \rightarrow (\, E\,) \\
E' \rightarrow & T' \rightarrow &
\end{array}
$$

- E.g., FIRST(F T’) = { id, num, (}
- We can use FIRST sets to determine which production to use!
  - Given nonterminal X, and all its productions
  - $X \rightarrow \gamma_1$, $X \rightarrow \gamma_2$, ... , $X \rightarrow \gamma_n$,
  - if FIRST(γ_1), ..., FIRST(γ_n) all mutually disjoint,
    then next character tells us which production to use

Computing FIRST Sets

- See Appel for algorithm. Intuition here...
- Consider FIRST(X Y Z)
- How do compute it? Do we just need to know FIRST(X)?
- What if X can derive the empty string?
- Then FIRST(Y) $\subseteq$ FIRST(X Y Z)
- What if Y can also derive the empty string?
- Then FIRST(Z) $\subseteq$ FIRST(X Y Z)

Computing FIRST, FOLLOW and Nullable

- To compute FIRST sets, we need to compute whether non-terminals can produce empty string
- FIRST(γ) = all terminal symbols that can start a string derived from γ
- Nullable(X) = true iff X can derive the empty string
- We will also compute:
  FOLLOW(X) = all terminals that can immediately follow X
  - i.e., t ∈ FOLLOW(X) if there is a derivation containing Xt
- Algorithm iterates computing these until fix point reached
- **Note:** knowing nullable(X) and FIRST(X) for all non-terminals X allows us to compute nullable(γ) and FIRST(γ) for arbitrary strings of symbols γ

Example

$S \rightarrow E$ eof

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow$ id

$F \rightarrow$ num

$F \rightarrow (E)$

	nullable	FIRST	FOLLOW
S	$\perp$		
E	$\perp$		
E'	$\top$		
T	$\perp$		
T'	$\top$		
F	$\perp$		

- X is nullable if there is a production X →γ where γ is empty, or γ is all nullable non-terminals
- T’ and E’ are nullable!
- And, we’ve finished nullable. Why?

Example

$S \rightarrow E\ \texttt{eof}$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow \texttt{id}$

$F \rightarrow \texttt{num}$

$F \rightarrow (\ E\ )$

	nullable	FIRST	FOLLOW
S	$\perp$		
E	$\perp$		
E'	$\top$	+ -	
T	$\perp$		
T'	$\top$	* /	
F	$\perp$	id num (	

- Given production $X \rightarrow t\gamma$, $t \in FIRST(X)$

Example

$S \rightarrow E$ eof

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow$ id

$F \rightarrow$ num

$F \rightarrow$ (E)

	nullable	FIRST	FOLLOW
S	$\perp$	id num (	
E	$\perp$	id num (	
E'	$\top$	+ -	
T	$\perp$	id num (	
T'	$\top$	* /	
F	$\perp$	id num (	

- Given production $X \rightarrow \gamma Y \sigma$,
  if nullable(γ) then FIRST(Y) $\subseteq$ FIRST(X)
- Repeat until no more changes...

Example

$S \to E$ eof

$E \to T\,E'$

$E' \to +\,T\,E'$

$E' \to -\,T\,E'$

$E' \to$

$T \to F\,T'$

$T' \to *\,F\,T'$

$T' \to /\,F\,T'$

$T' \to$

$F \to$ id

$F \to$ num

$F \to (\,E\,)$

	nullable	FIRST	FOLLOW
S	$\perp$	id num (	
E	$\perp$	id num (	eof)
E'	$\top$	+ -	eof)
T	$\perp$	id num (	+ - eof)
T'	$\top$	* /	+ - eof)
F	$\perp$	id num (	* / + - eof)

- Given production X → γZδσ

FIRST(δ) ⊆ FOLLOW(Z)

  – and if δ is nullable then FIRST(σ) ⊆FOLLOW(Z)
  – and if δσ is nullable then FOLLOW(X) ⊆FOLLOW(Z

Predictive Parsing Table

- Make **predictive parsing table** with rows non-terminals, columns terminals
  - Table entries are productions
  - When parsing nonterminal X, and next token is t, entry for X and t will tell us which production to use

Example

$S \rightarrow E$ eof

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow$ id

$F \rightarrow$ num

$F \rightarrow (\ E\ )$

	nullable	FIRST	FOLLOW
S	$\perp$	id num (	
E	$\perp$	id num (	eof)
E'	$\top$	+ -	eof)
T	$\perp$	id num (	+ - eof)
T'	$\top$	* /	+ - eof)
F	$\perp$	id num (	* / + - eof)

	id	num	+	-	*	/	(	)	eof
S	$S \rightarrow E$ eof	$S \rightarrow E$ eof					$S \rightarrow E$ eof		
E	$E \rightarrow T\ E'$	$E \rightarrow T\ E'$					$E \rightarrow T\ E'$		
E'			$E' \rightarrow +\ T\ E'$	$E' \rightarrow -\ T\ E'$				$E' \rightarrow$	$E' \rightarrow$
T	$T \rightarrow F\ T'$	$T \rightarrow F\ T'$					$T \rightarrow F\ T'$		
T'			$T' \rightarrow$	$T' \rightarrow$	$T' \rightarrow *\ F\ T'$	$T' \rightarrow /\ F\ T'$		$T' \rightarrow$	$T' \rightarrow$
F	$F \rightarrow$ id	$F \rightarrow$ num					$F \rightarrow (\ E\ )$		

Example

$S \rightarrow E\ \texttt{eof}$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow \texttt{id}$

$F \rightarrow \texttt{num}$

$F \rightarrow (\ E\ )$

	id	num	+	-	*	/	(	)	eof
S	$S \rightarrow E\ \texttt{eof}$	$S \rightarrow E\ \texttt{eof}$					$S \rightarrow E\ \texttt{eof}$		
E	$E \rightarrow T\ E'$	$E \rightarrow T\ E'$					$E \rightarrow T\ E'$		
E'			$E' \rightarrow +\ T\ E'$	$E' \rightarrow -\ T\ E'$				$E' \rightarrow$	$E' \rightarrow$
T	$T \rightarrow F\ T'$	$T \rightarrow F\ T'$					$T \rightarrow F\ T'$		
T'			$T' \rightarrow$	$T' \rightarrow$	$T' \rightarrow *\ F\ T'$	$T' \rightarrow /\ F\ T'$		$T' \rightarrow$	$T' \rightarrow$
F	$F \rightarrow \texttt{id}$	$F \rightarrow \texttt{num}$					$F \rightarrow (\ E\ )$		

- If each cell contains at most one production, ***parsing is predictive***!
  - **Table tells us exactly which production to apply**

`( foo + 7 ) eof`

$S \to E$ eof

$E \to T\, E'$

$E' \to +\, T\, E'$

$E' \to -\, T\, E'$

$E' \to$

$T \to F\, T'$

$T' \to *\, F\, T'$

$T' \to /\, F\, T'$

$T' \to$

$F \to$ id

$F \to$ num

$F \to (E)$

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\, E'$	$E \to T\, E'$					$E \to T\, E'$		
E'			$E' \to +\, T\, E'$	$E' \to -\, T\, E'$				$E' \to$	$E' \to$
T	$T \to F\, T'$	$T \to F\, T'$					$T \to F\, T'$		
T'			$T' \to$	$T' \to$	$T' \to *\, F\, T'$	$T' \to /\, F\, T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

- If each cell contains at most one production, ***parsing is predictive***!
  - **Table tells us exactly which production to apply**

(foo + 7) eof

S	Parse S, next token is (, use $S \to E$ eof
E eof	Parse E, next token is (, use $E \to T\, E'$
$T\, E'$ eof	Parse T, next token is (, use $T \to F\, T'$
$F\, T'\, E'$ eof	Parse F, next token is (, use $F \to (E)$
$(E)\, T'\, E'$ eof	Parse E, next token is id, use $E \to T\, E'$
$(T\, E')\, T'\, E'$ eof	Parse T, next token is id, use $T \to F\, T'$
$(F\, T'\, E')\, T'\, E'$ eof	Parse F, next token is id, use $F \to$ id

$S \to E$ eof

$E \to T\ E'$
$E' \to +\ T\ E'$
$E' \to -\ T\ E'$
$E' \to$

$T \to F\ T'$
$T' \to *\ F\ T'$
$T' \to /\ F\ T'$
$T' \to$

$F \to$ id
$F \to$ num
$F \to (\ E\ )$

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\ E'$	$E \to T\ E'$					$E \to T\ E'$		
E'			$E' \to +\ T\ E'$	$E' \to -\ T\ E'$				$E' \to$	$E' \to$
T	$T \to F\ T'$	$T \to F\ T'$					$T \to F\ T'$		
T'			$T' \to$	$T' \to$	$T' \to *\ F\ T'$	$T' \to /\ F\ T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (\ E\ )$		

- If each cell contains at most one production, ***parsing is predictive***!
  - **Table tells us exactly which production to apply**

(foo + 7) eof

$(\ F\ T'\ E'\ )\ T'\ E'$ eof	Parse F, next token is id, use $F \to$ id
$($ id $T'\ E'\ )\ T'\ E'$ eof	Parse T', next token is +, use $T' \to$
$($ id $E'\ )\ T'\ E'$ eof	Parse E', next token is +, use $E' \to +\ T\ E'$
$($ id $+\ T\ E'\ )\ T'\ E'$ eof	Parse T, next token is num, use $T \to F\ T'$
$($ id $+\ F\ T'\ E'\ )\ T'\ E'$ eof	Parse F, next token is num, use $F \to$ num

$S \to E$ eof

$E \to T\ E'$

$E' \to +\ T\ E'$

$E' \to -\ T\ E'$

$E' \to$

$T \to F\ T'$

$T' \to *\ F\ T'$

$T' \to /\ F\ T'$

$T' \to$

$F \to$ id

$F \to$ num

$F \to (E)$

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\ E'$	$E \to T\ E'$					$E \to T\ E'$		
E'			$E' \to +\ T\ E'$	$E' \to -\ T\ E'$				$E' \to$	$E' \to$
T	$T \to F\ T'$	$T \to F\ T'$					$T \to F\ T'$		
T'			$T' \to$	$T' \to$	$T' \to *\ F\ T'$	$T' \to /\ F\ T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

- If each cell contains at most one production, ***parsing is predictive***!
  - **Table tells us exactly which production to apply**

(foo + 7) eof

(id + $F\ T'\ E'$) $T'\ E'$ eof	Parse F, next token is num, use $F \to$ num
(id + num $T'\ E'$) $T'\ E'$ eof	Parse T', next token is), use $T' \to$
(id + num E') $T'\ E'$ eof	Parse E', next token is), use $E' \to$
(id + num) $T'\ E'$ eof	Parse T', next token is eof, use $T' \to$
(id + num) E' eof	Parse E', next token is eof, use $E' \to$
(id + num) eof	

LL(1), LL(k), LL(*)

- Grammars whose predictive parsing table contain at most one production per cell are called LL(1)

LL(1)

Left-to-right parse

i.e., go through token stream from left to right. (Almost all parsers do this)

Leftmost derivation

Derivation expands the leftmost non-terminal

1-symbol lookahead

LL(1), LL(k), LL(*)

- Grammars whose predictive parsing table contain at most one production per cell are called LL(1)
  - Can be generalized to LL(2), LL(3), etc.
    - Columns of predictive parsing table have k tokens
    - FIRST(X) generalized to FIRST-k(X)
- An LL(*) grammar can determine next production using finite (but maybe unbounded) lookahead
- An ambiguous grammar is not LL(k) for any k, or even LL(*)

LR(k)

- What if grammar is unambiguous but not LL(k)?
- LR(k) parsing is more powerful technique

LR(k)

Left-to-right parse

i.e., go through token stream from left to right. (Almost all parsers do this)

Rightmost derivation

Derivation expands the rightmost non-terminal **(Constructs derivation in reverse order!)**

k-symbol lookahead

LR(k)

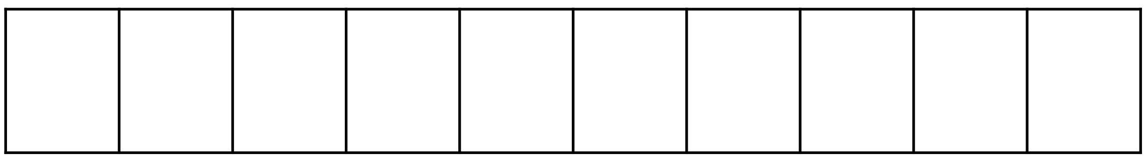
- **Basic idea: LR parser has a stack and input**

- Given contents of stack and k tokens look-ahead parser does one of following operations:
  - Shift: move first input token to top of stack
  - Reduce: top of stack matches rule, e.g., $\boldsymbol{X \rightarrow A\ B\ C}$
    - Pop C, pop B, pop A, and push X

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E)$$
$$E \rightarrow E + E$$

Stack



Input

(3 + 4) + (5 + 6)

Example

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E)$$
$$E \rightarrow E + E$$

Stack

(	E								

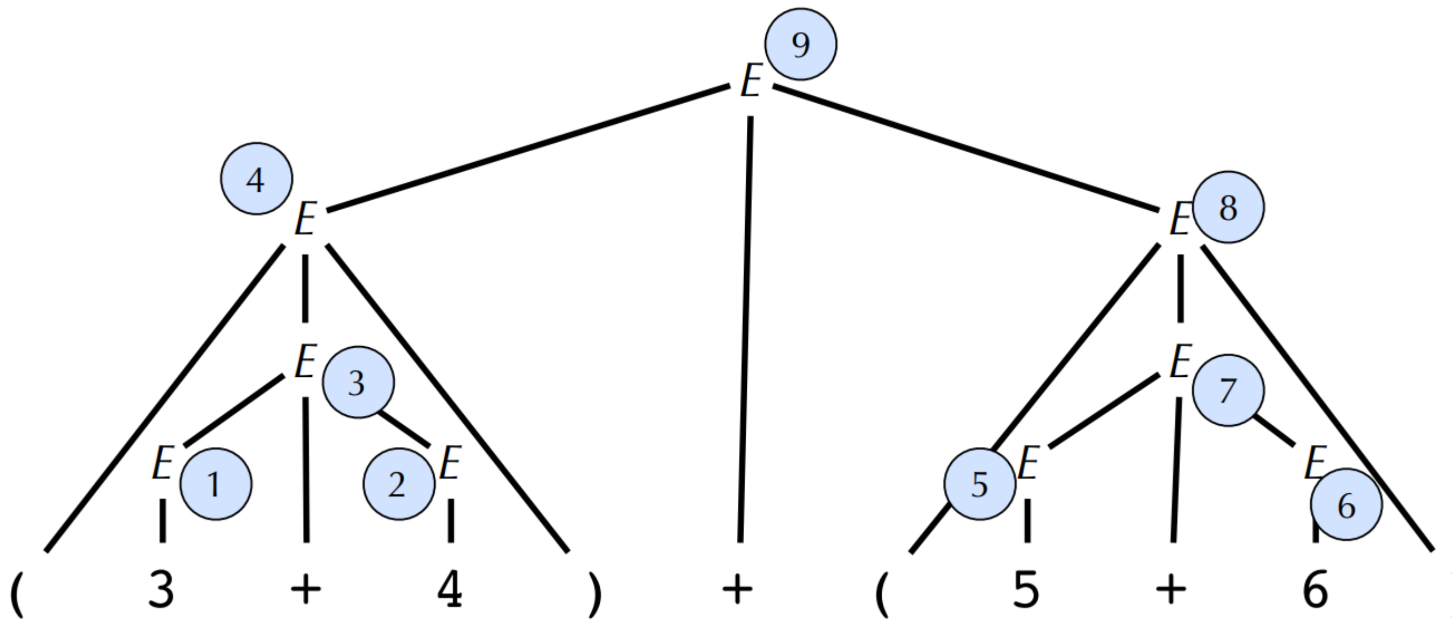
Input

(3 + 4) + (5 + 6)

Shift (on to stack
Shift 3 on to stack
Reduce using rule $E \rightarrow \texttt{int}$
Shift + on to stack

Rightmost derivation

- LR parsers produce a rightmost derivation

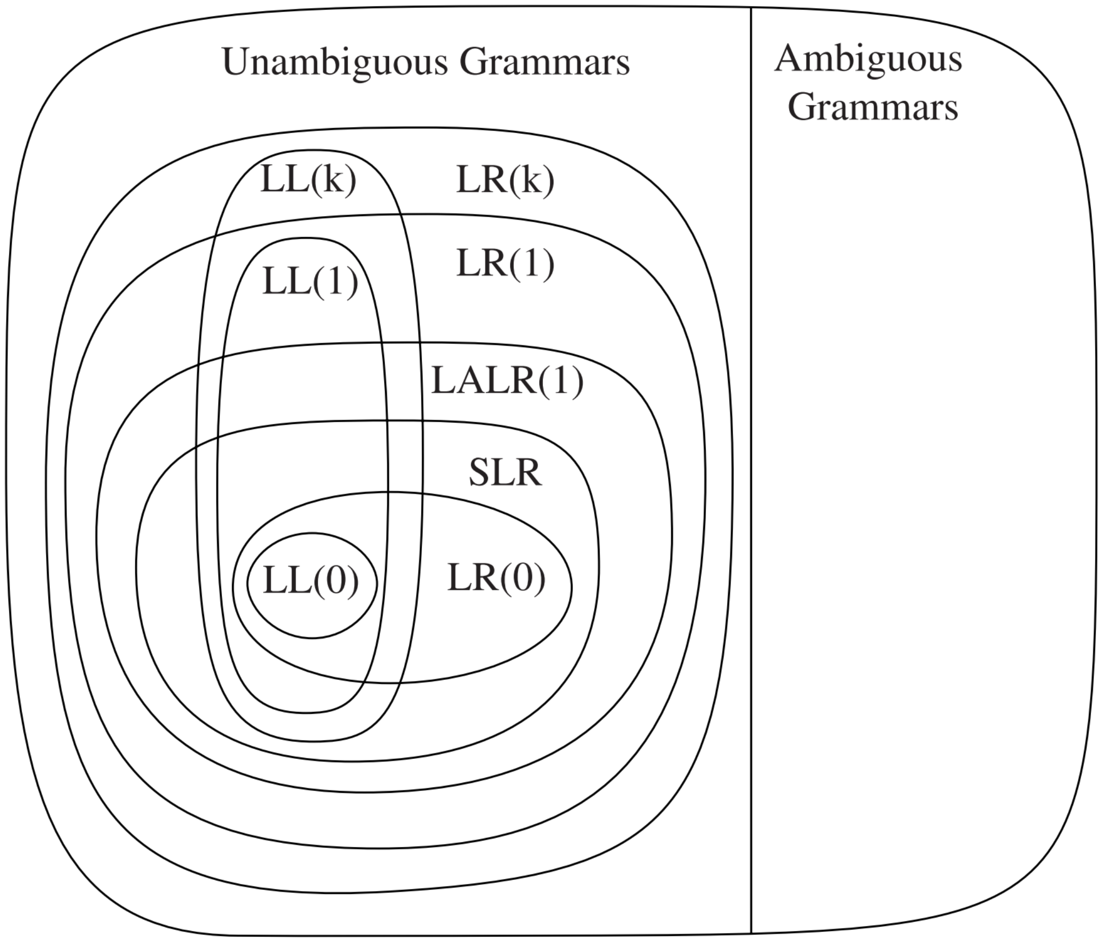


- But do reductions in reverse order

What Action to Take?

- How does the LR(k) parser know when to shift and to reduce?
- Uses a DFA
  - At each step, parser runs DFA using symbols on stack as input
    - Input is sequence of terminals and non-terminals from bottom to top
  - Current state of DFA plus next k tokens indicate whether to shift or reduce

Hierarchy of Grammar Classes



- In practice, LR(1) is used for LR parsing
  - not LR(0) or LR(k) for k>1
- LALR: Look-Ahead LR parser

Example

- LR grammar that cannot be represented by LL

```
list ::= list ',' element | element
```

  - This grammar can be easily converted to LL grammars by left-factoring or eliminations of left-recursion

- Languages that are LR but not LL: (YES, dangling if-else has the same issue!)
  - there exists LR(1) grammar, but no LL(k) grammar (for any k)!
  - strings that can satisfy the following pattern: $\{a^i b^j \mid i>=j\}$

```
S ::= a S | P
P ::= a P b | ε
```