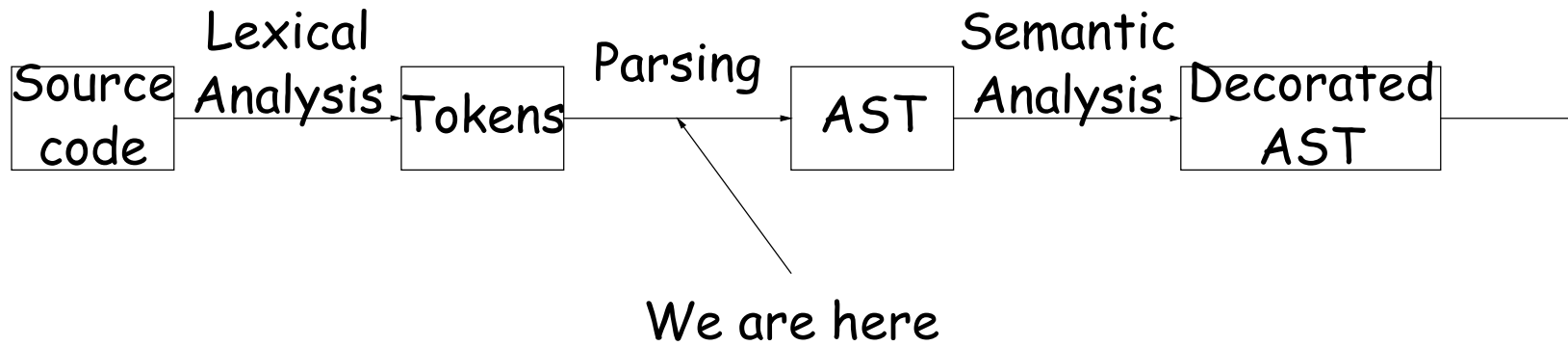


# Lecture 6: Parsing

# A Glance at the Map



# Review: BNF

- BNF is another pattern-matching language;
- Alphabet typically set of *tokens*, such as from lexical analysis, referred to as *terminal symbols* or *terminals*.
- Matching rules have form:

$$X : \alpha_1 \alpha_2 \cdots \alpha_n,$$

where  $X$  is from a set of *nonterminal symbols* (or *nonterminals* or *meta-variables*),  $n \geq 0$ , and each  $\alpha_i$  is a terminal or nonterminal symbol.

- For emphasis, may write  $X : \epsilon$  when  $n = 0$ .
- Read  $X : \alpha_1 \alpha_2 \cdots \alpha_n$ , as  
    "An  $X$  may be formed from the concatenation of an  $\alpha_1, \alpha_2, \dots, \alpha_n$ ."- Designate one nonterminal as the *start symbol*.
- Set of all matching rules is a *context-free grammar*.

# Derivations

- String (of terminals)  $T$  is in the language described by grammar  $G$ , ( $T \in L(G)$ ) if there is a *derivation of  $T$*  from the start symbol of  $G$ .
- Derivation of  $T = \tau_1 \cdots \tau_k$  from nonterminal  $A$  is sequence of *sentential forms*:

$$A \Rightarrow \alpha_{11}\alpha_{12} \dots \Rightarrow \alpha_{21}\alpha_{22} \dots \Rightarrow \dots \Rightarrow \tau_1 \dots \tau_k$$

where each  $\alpha_{ij}$  is a terminal or nonterminal symbol.

- We say that

$$\alpha_1 \cdots \alpha_{m-1} B \alpha_{m+1} \cdots \alpha_n \Rightarrow \alpha_1 \cdots \alpha_{m-1} \beta_1 \cdots \beta_p \alpha_{m+1} \cdots \alpha_n$$

if  $B : \beta_1 \cdots \beta_p$  is a production. ( $1 \leq m \leq n$ ).

- If  $\Phi$  and  $\Phi'$  are sentential forms, then  $\Phi_1 \xRightarrow{*} \Phi_2$  means that 0 or more  $\Rightarrow$  steps turns  $\Phi_1$  into  $\Phi_2$ .  $\Phi_1 \xRightarrow{+} \Phi_2$  means 1 or more  $\Rightarrow$  steps does it.
- So if  $S$  is start symbol of  $G$ , then  $T \in L(G)$  iff  $S \xRightarrow{+} T$ .

# Example of Derivation

1.  $e : s \text{ ID}$
2.  $e : s \text{ ' ( ' e ' ) '}$
3.  $e : e \text{ ' / ' e}$
4.  $s :$
5.  $s : \text{'+'}$
6.  $s : \text{'-'}$

## Alternative Notation

$e : s \text{ ID}$   
 $\quad | s \text{ ' ( ' e ' ) '}$   
 $\quad | e \text{ ' / ' e}$   
 $s : \epsilon \mid \text{'+'} \mid \text{'-'}$

**Problem:** Derive  $- \text{ ID } / ( \text{ ID } / \text{ ID } )$

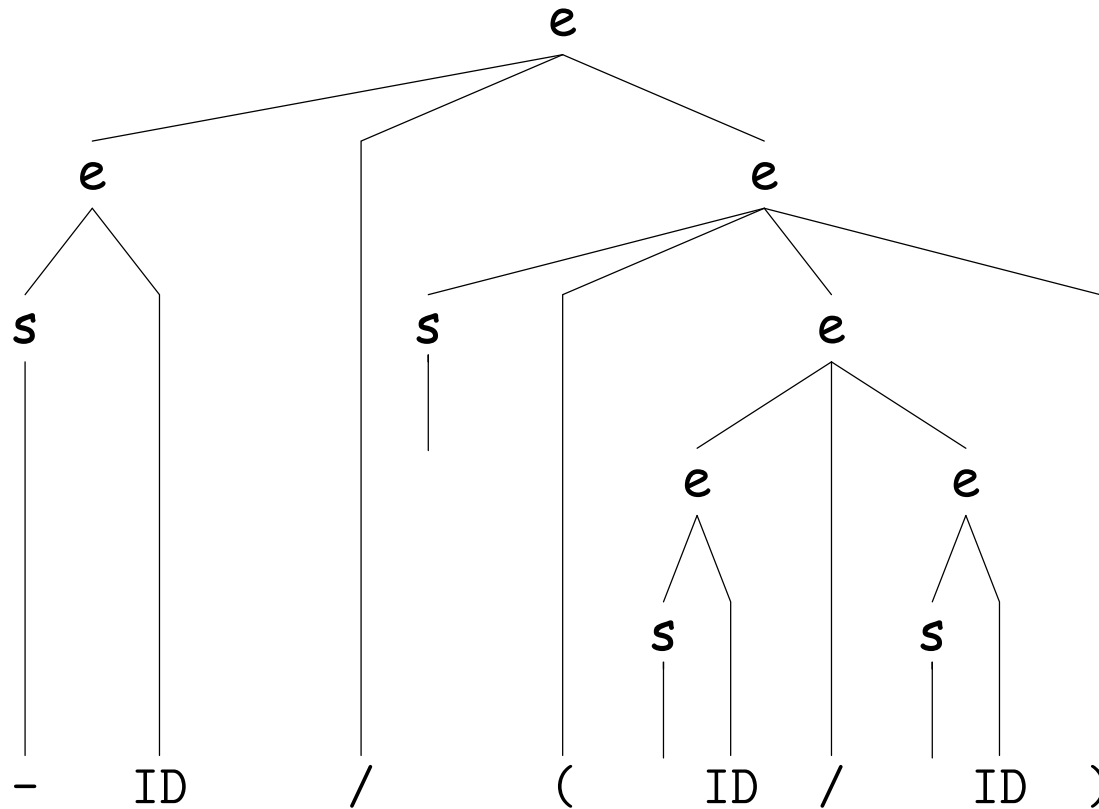
$$\begin{aligned}
 e &\xrightarrow{3} e / e \xrightarrow{1} s \text{ ID } / e \xrightarrow{6} - \text{ ID } / e \xrightarrow{2} - \text{ ID } / s ( e ) \\
 &\xrightarrow{4} - \text{ ID } / ( e ) \xrightarrow{3} - \text{ ID } / ( e / e ) \xrightarrow{1} - \text{ ID } / ( s \text{ ID } / e ) \\
 &\xrightarrow{4} - \text{ ID } / ( \text{ ID } / e ) \xrightarrow{1} - \text{ ID } / ( \text{ ID } / s \text{ ID } ) \\
 &\xrightarrow{4} - \text{ ID } / ( \text{ ID } / \text{ ID } )
 \end{aligned}$$

# Types of Derivation

- *Context free* means can replace nonterminals in any order (i.e., regardless of context) to get same result (as long as you use same productions).
- So, if we use a particular rule for selecting nonterminal to “produce” from, can characterize derivation by just listing productions.
- Previous example was *leftmost derivation*: always choose leftmost nonterminals. Completely characterized by list of productions: 3, 1, 6, 2, 4, 3, 1, 4, 1, 4.

# Derivations and Parse Trees

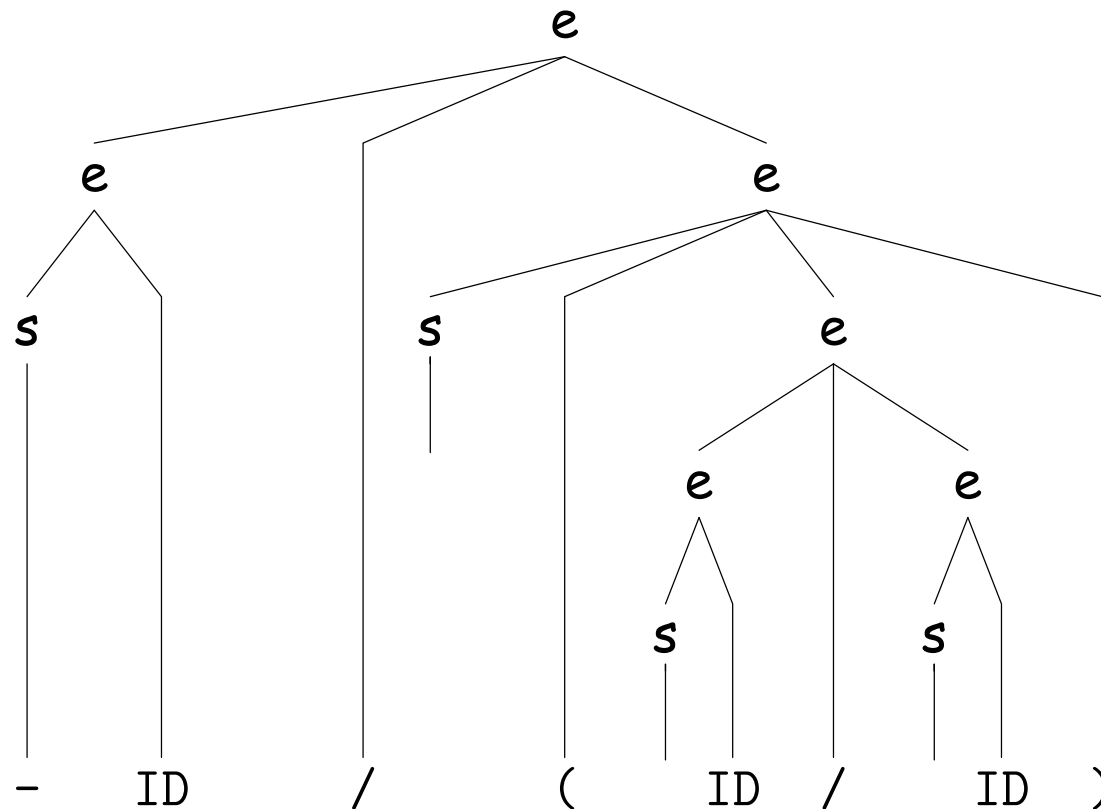
- A leftmost derivation also completely characterized by parse tree:



- What is the rightmost derivation for this?

# Derivations and Parse Trees

- A leftmost derivation also completely characterized by parse tree:



- What is the rightmost derivation for this?

$$\begin{aligned}
 e &\xRightarrow{3} e / e \xRightarrow{2} e / s ( e ) \xRightarrow{3} e / s ( e / e ) \\
 &\xRightarrow{1} e / s ( e / s ID ) \xRightarrow{4} e / s ( e / ID ) \\
 &\xRightarrow{1} e / s ( s ID / ID ) \xRightarrow{4} e / s ( ID / ID ) \\
 &\xRightarrow{4} e / ( ID / ID ) \xRightarrow{1} s ID / ( ID / ID ) \xRightarrow{6} - ID / ( ID / ID )
 \end{aligned}$$

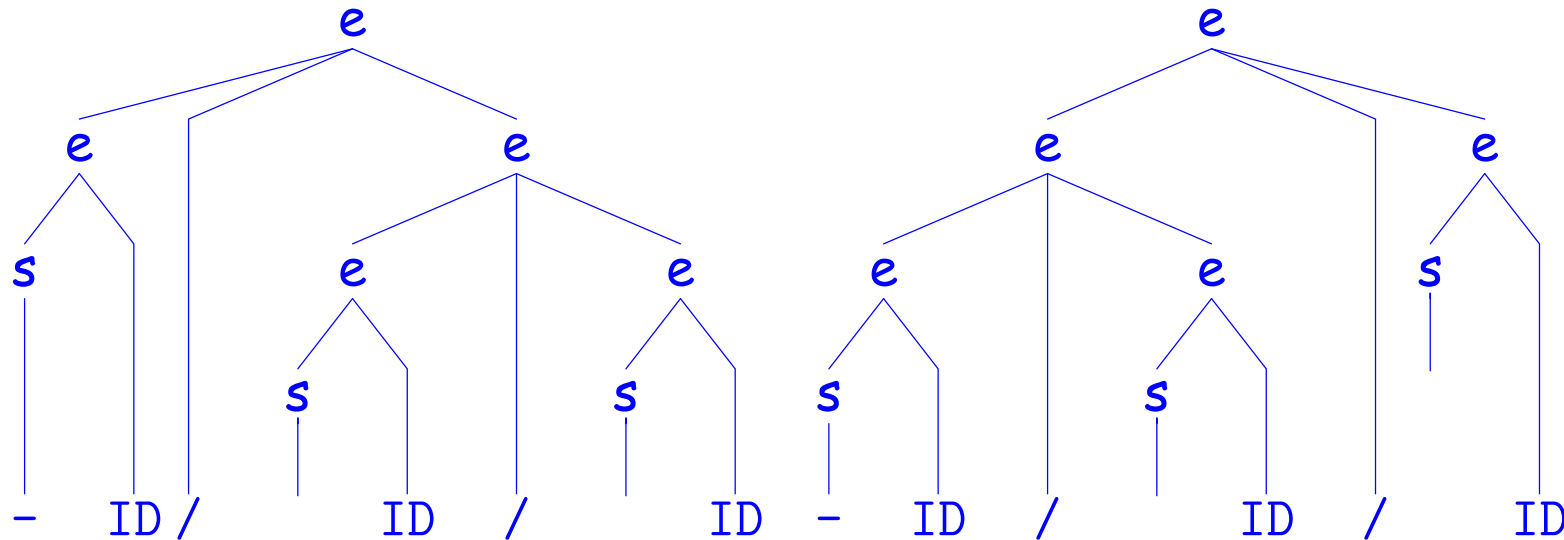


# Ambiguity

- [illegible]

# Ambiguity

- Only one derivation for previous example.
- What about 'ID / ID / ID'?
- Claim there are two parse trees, corresponding to two leftmost derivations. What are they?



- If there exists even one string like ID / ID / ID in  $L(G)$ , we say  $G$  is ambiguous (even if other strings only have one parse tree).

# Review: Syntax-Directed Translation

- Want the structure of sentences, not just whether they are in the language, because this drives translation.
- Associate translation rules to each production, just as Flex associated actions with matching patterns.
- Bison notation:

`e : e '/' e { $$ = doDivide($1, $3); }`

provides way to refer to and set *semantic values* on each node of a parse tree.

- Compute these semantic values from leaves up the parse tree.
- Same as the order of a *rightmost derivation in reverse* (a.k.a a *canonical derivation*).
- Alternatively, just perform arbitrary actions in the same order.

## Example: Conditional statement

**Problem:** `if-else` or `if-elif-else` statements in Python (`else` optional). Assume that only (indented) suites may be used for then and else clauses, that nonterminal `stmt` defines an individual statement (one per line), and that nonterminal `expr` defines an expression. Lexer supplies `INDENTs` and `DEDENTs`. A `cond` is a kind of `stmt`.

## Example: Conditional statement

**Problem:** `if-else` or `if-elif-else` statements in Python (`else` optional). Assume that only (indented) suites may be used for then and else clauses, that nonterminal `stmt` defines an individual statement (one per line), and that nonterminal `expr` defines an expression. Lexer supplies `INDENT`s and `DEDENT`s. A `cond` is a kind of `stmt`.

```
expr : ...
stmt : ... | cond | ...
cond : "if" expr ':' suite elifs else
suite: INDENT stmts DEDENT
stmts: stmt | stmts stmt
elifs:  $\epsilon$  | "elif" expr ':' suite elifs
else :  $\epsilon$  | "else" ':' suite
```

## Example: Conditional statement in Java

**Problem:** `if-else` in Java. Assume that nonterminal `stmt` defines an individual statement (including a block in `{}`).

## Example: Conditional statement in Java

**Problem:** `if-else` in Java. Assume that nonterminal `stmt` defines an individual statement (including a block in `{}`).

```
expr : ...  
stmt : ... | cond | ...  
cond : "if" '(' expr ')' stmt else  
else :  $\epsilon$  | "else" stmt
```

But this doesn't quite work: recognizes correct statements and rejects incorrect ones, but is ambiguous. E.g.,

```
if (foo) if (bar) walk(); else chewGum();
```

Do we chew gum if `foo` is false? That is, is this equivalent to

```
if (foo) { if (bar) walk(); } else chewGum();  
/*or*/ if (foo) { if (bar) walk(); else chewGum(); } ?
```

## Example resolved: Conditional statement in Java

The rule is supposed to be "each 'else' attaches to the nearest open 'if' on the left," which is captured by:

```
expr : ...
stmt : ... | cond | ...
stmt_closed : ... | cond_closed | ...
cond_closed : "if" '(' expr ')' stmt_closed "else" stmt_closed
cond : "if" '(' expr ')' stmt
      | "if" '(' expr ')' stmt_closed "else" stmt
```

This does not allow us to interpret

```
if (foo) if (bar) walk(); else chewGum();
```

as

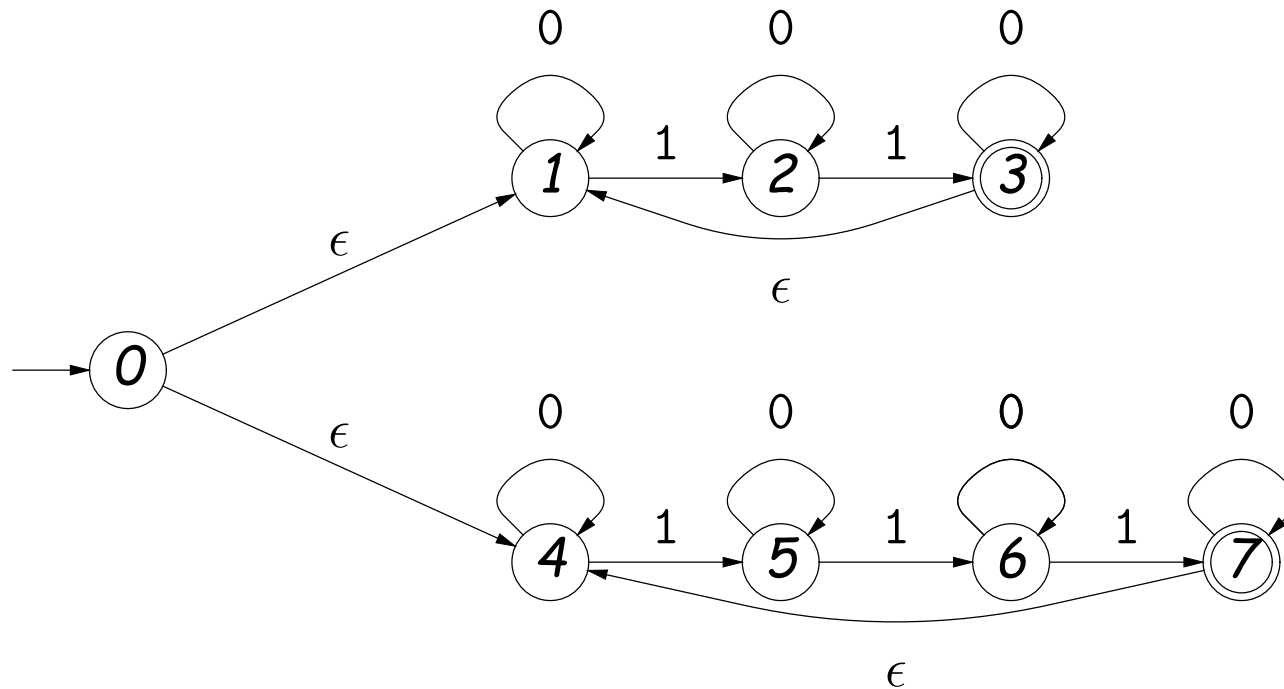
```
if (foo) { if (bar) walk(); } else chewGum();
```

But it's not exactly clear, is it?



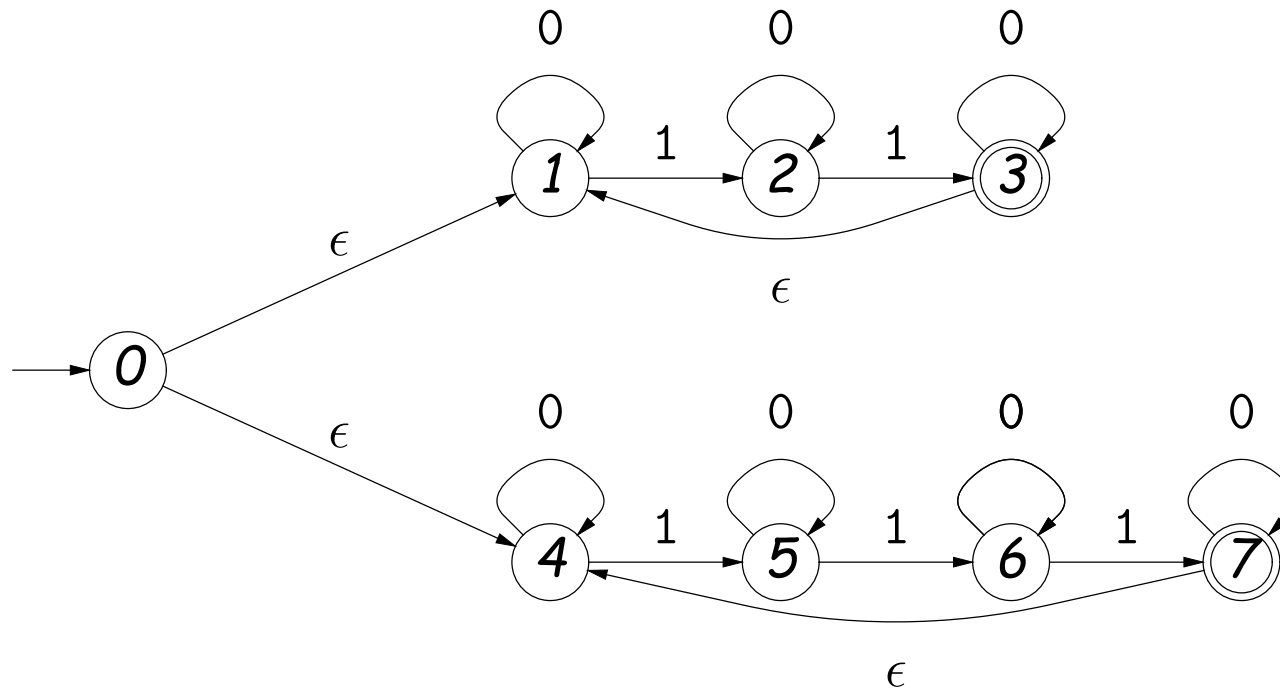
## Puzzle: NFA to BNF

**Problem:** What BNF grammar accepts the same string as this NFA?



# Puzzle: NFA to BNF

**Problem:** What BNF grammar accepts the same string as this NFA?



A conventional answer (adapted from class):

$S: S2s \mid S3s$

$Z: '0' Z \mid \epsilon$

$S2s: S2 \mid S2 S2s$

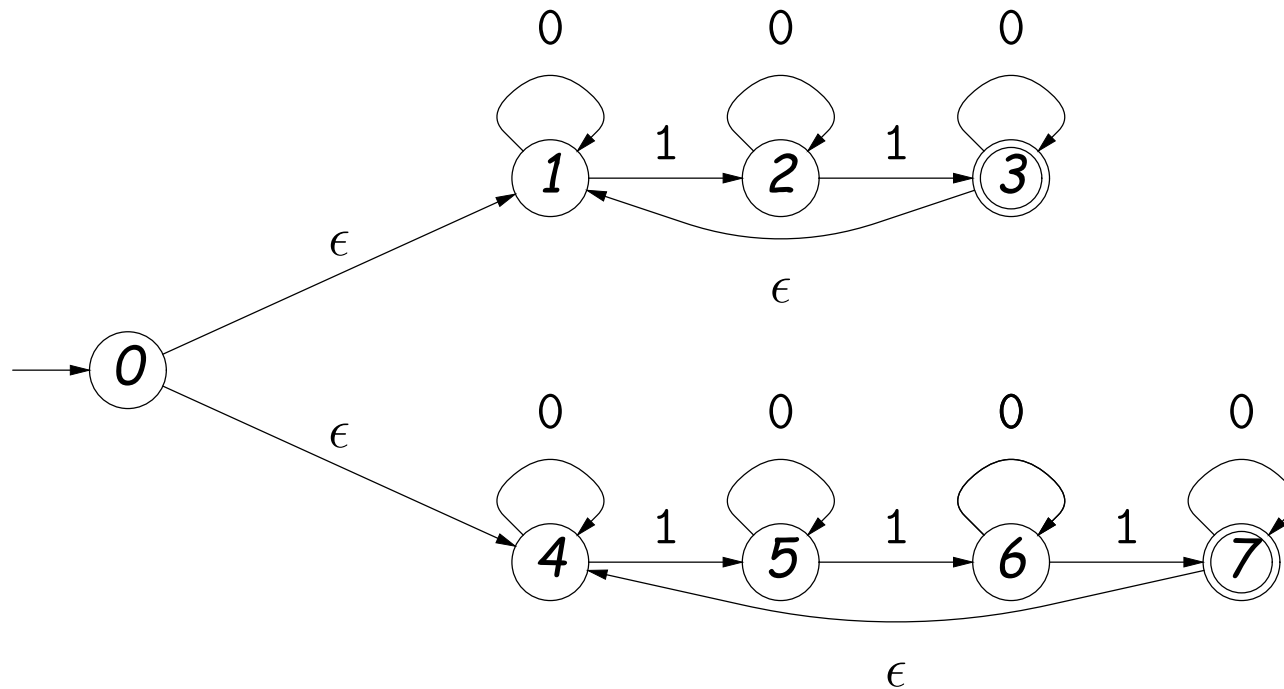
$S3s: S3 \mid S3 S3s$

$S2: Z '1' Z '1' Z$

$S3: Z '1' Z '1' Z '1' Z$

# Puzzle: NFA to BNF

**Problem:** What BNF grammar accepts the same string as this NFA?



General answer (adaptable to any NFA), with one nonterminal per state:

$S_0: S_1 \mid S_4$

$S_1: '1' S_2 \mid '0' S_1$

$S_2: '1' S_3 \mid '0' S_2$

$S_3: S_1 \mid '0' S_3 \mid \epsilon$

$S_4: '1' S_5 \mid '0' S_4$

$S_5: '1' S_6 \mid '0' S_5$

$S_6: '1' S_7 \mid '0' S_6$

$S_7: S_4 \mid '0' S_7 \mid \epsilon$

Nonterminal  $S_k$  is "the set of strings that will get me from  $S_k$  in the NFA to a final state in the NFA."