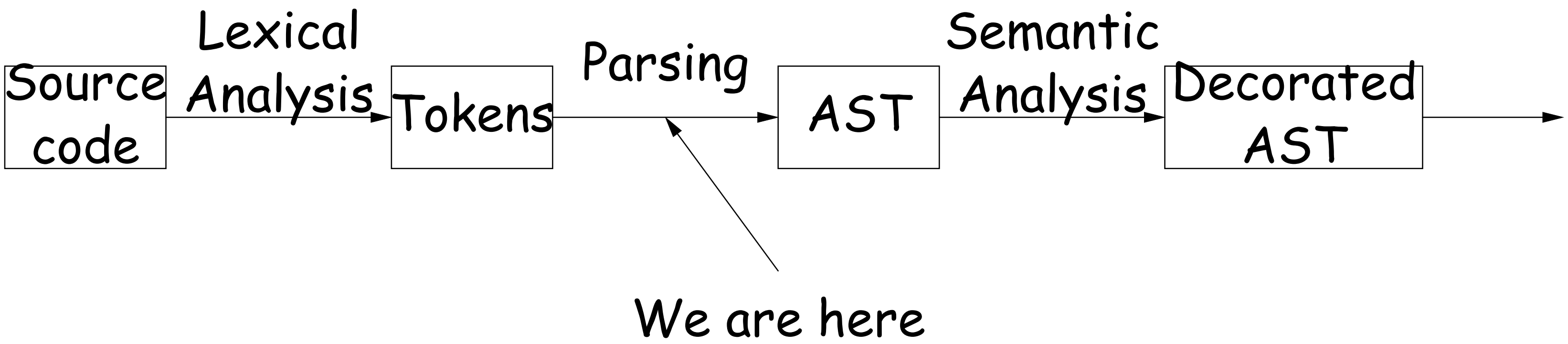
Lecture 4: Parsing

A Glance at the Map



Limitations of Finite Automata and Regular Expressions

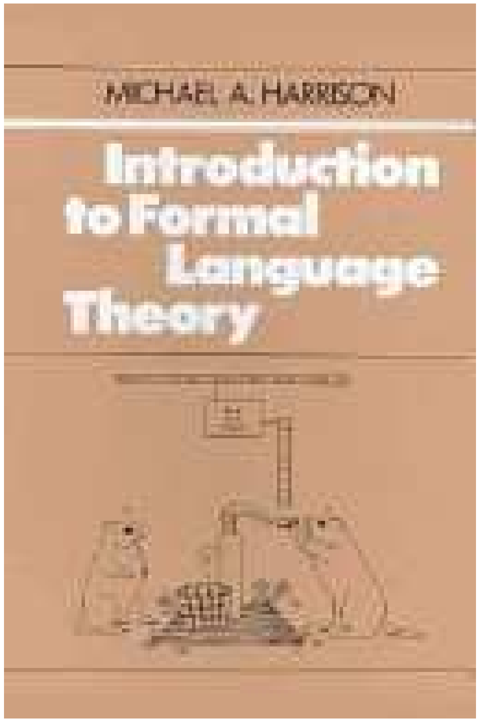
- Why not use regular expressions to describe other parts of a language's grammar, and finite automata to process it?
- Regular languages (what finite automata can recognize) cannot handle common features of current programming languages.
- Simple example: recognize the language $L = \{a^n b^n | n \geq 0\}$ (where a^n means a string consisting of a sequence of n '$\texttt{a}$'s).
- Consider a DFSA, M, that supposedly recognizes L. We feed it each of the strings a^n for $n \geq 0$. After reading the string a^k, M will be in some state, call it s_k.
- For M to work, s_k must be different for each different k. Why?

Limitations of Finite Automata and Regular Expressions

- Why not use regular expressions to describe other parts of a language's grammar, and finite automata to process it?
- Regular languages (what finite automata can recognize) cannot handle common features of current programming languages.
- Simple example: recognize the language $L = \{a^n b^n | n \geq 0\}$ (where a^n means a string consisting of a sequence of n '$\mathtt{a}$'s).
- Consider a DFSA, M, that supposedly recognizes L. We feed it each of the strings a^n for $n \geq 0$. After reading the string a^k, M will be in some state, call it s_k.
- For M to work, s_k must be different for each different k. Why?

  Proof: Suppose that after reading either a^k or $a^{k'}$, M is in state s_k, $k \neq k'$. Starting from s_k, reading b^k must then put M in a final state. That means that M will end up in a final state for both $a^k b^k$ and $a^{k'} b^k$. But that would be an error. Therefore that would require an infinite set of states. What part of "finite state" do you not understand?

The Pumping Lemma



More generally, we have the Pumping Lemma for Regular Languages: For every regular language L, there must be $p \geq 1$ such that every string, $s \in L$, of length $\geq p$ has the form xyz (concatenation of three strings) such that

- $|y| \geq 1$
- $|xy| \leq p$
- $xy^nz \in L$ for every $n \geq 0$.

Thus, every sufficiently large s in L can be "pumped up" with arbitrarily many copies of some non-empty substring of s.

Nerdy Question: When does this does *not* imply that there are any "pumpable" strings in L?

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 α_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 α_1, α_2, ..., α_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:*

$$A \;\Rightarrow\; \alpha_{11}\alpha_{12}\ldots \;\Rightarrow\; \alpha_{21}\alpha_{22}\ldots \;\Rightarrow\; \cdots \;\Rightarrow\; \tau_1\ldots\tau_k$$

where each α_{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. ($1 \le m \le n$).

- If Φ and Φ' are sentential forms, then $\Phi_1 \stackrel{*}{\Longrightarrow} \Phi_2$ means that 0 or more $\Rightarrow$ steps turns Φ_1 into Φ_2. $\Phi_1 \stackrel{+}{\Longrightarrow} \Phi_2$ means 1 or more $\Rightarrow$ steps does it.
- So if S is start symbol of G, then $T \in L(G)$ iff $S \stackrel{+}{\Longrightarrow} T$.

Example of Derivation

1. `e ::= s ID`
2. `e ::= s '(' e ')'`
3. `e ::= e '/' e`
4. `s ::=`
5. `s ::= '+'`
6. `s ::= '-'`

Alternative Notation

```
e ::= s ID
   | s '(' e ')'
   | e '/' e
s ::= ε | '+' | '-'
```

Problem: Derive `- ID / ( ID / ID )`

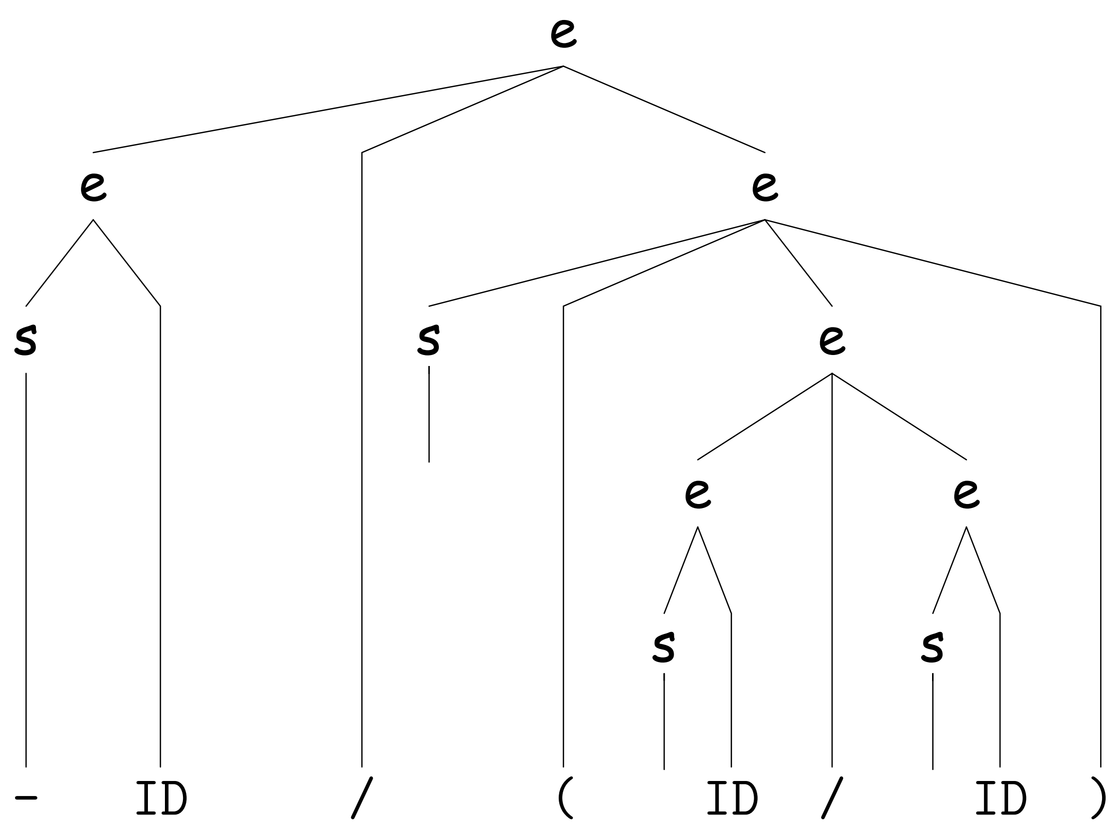
$$\texttt{e} \stackrel{3}{\Longrightarrow} \texttt{e / e} \stackrel{1}{\Longrightarrow} \texttt{s ID / e} \stackrel{6}{\Longrightarrow} \texttt{- ID / e} \stackrel{2}{\Longrightarrow} \texttt{- ID / s (e)}$$
$$\stackrel{4}{\Longrightarrow} \texttt{- ID / (e)} \stackrel{3}{\Longrightarrow} \texttt{- ID / (e / e)} \stackrel{1}{\Longrightarrow} \texttt{- ID / (s ID / e)}$$
$$\stackrel{4}{\Longrightarrow} \texttt{- ID / (ID / e)} \stackrel{1}{\Longrightarrow} \texttt{- ID / (ID / s ID)}$$
$$\stackrel{4}{\Longrightarrow} \texttt{- ID / (ID / ID)}$$

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

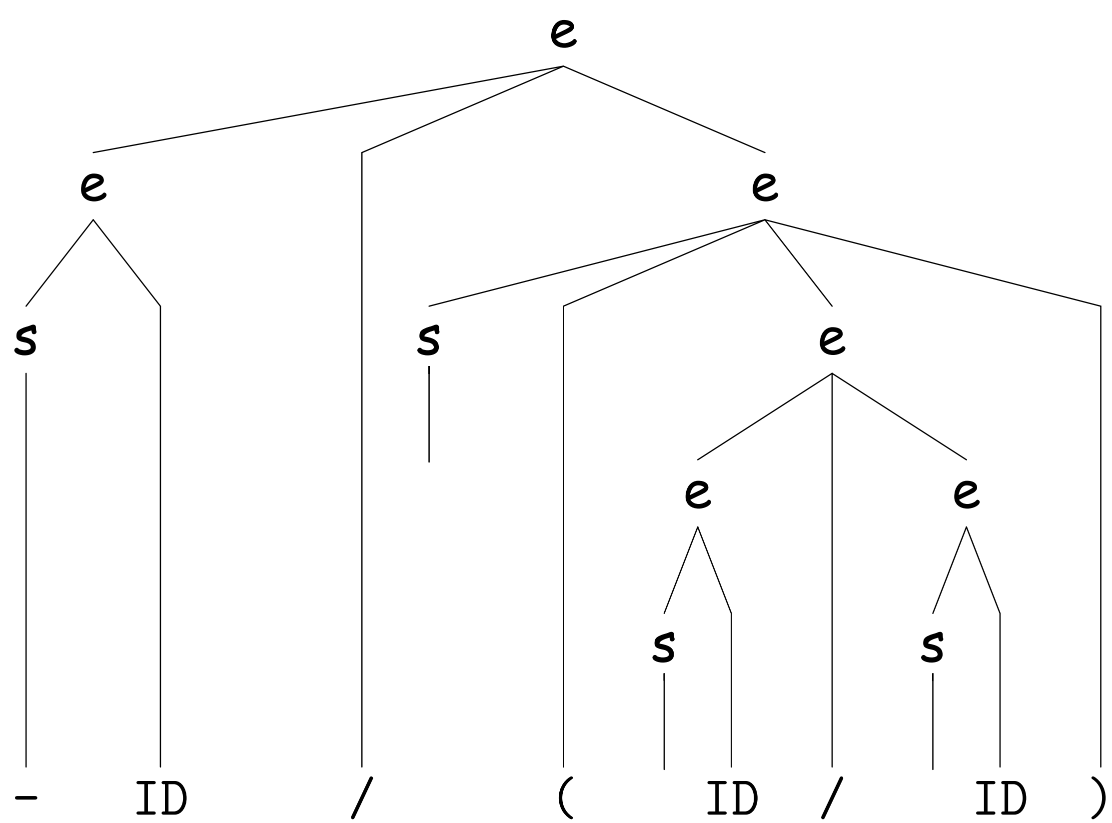
- A leftmost derivation also completely characterized by parse tree:



- 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?

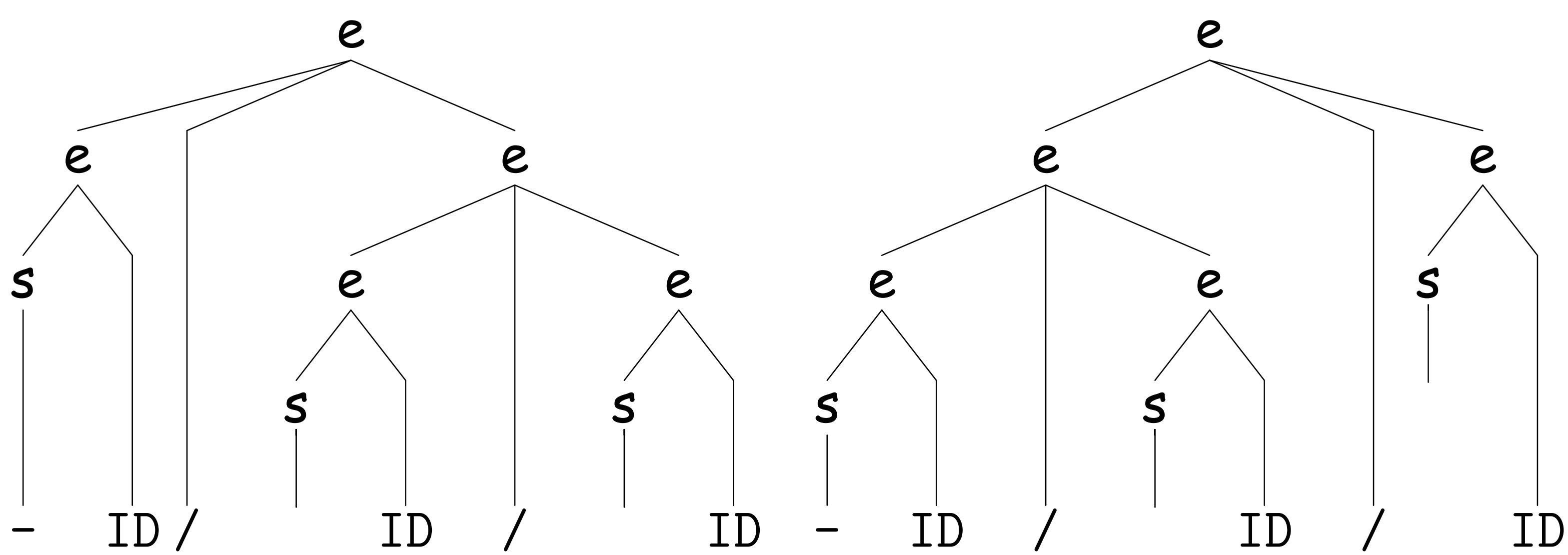
$$
\begin{aligned}
\texttt{e} &\stackrel{3}{\Longrightarrow} \texttt{e / e} \stackrel{2}{\Longrightarrow} \texttt{e / s (e)} \stackrel{3}{\Longrightarrow} \texttt{e / s (e / e)}\\
&\stackrel{1}{\Longrightarrow} \texttt{e / s (e / s ID)} \stackrel{4}{\Longrightarrow} \texttt{e / s (e / ID)}\\
&\stackrel{1}{\Longrightarrow}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

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).

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.
- CUP notation:

```
e ::= e:left '/' e:right  {: RESULT = doDivide(left, right); :}
```

  provides way to refer to and set *semantic values* on each node of a parse tree. Here, the names after the colons—`left` and `right`—stand for the semantic values of the phrases matched by the two instances of the nonterminal 'e', and `RESULT` stands for the semantic value of the nonterminal 'e' on the left.
- 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 INDENTs and DEDENTs. 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 ::= ϵ    | "elif" expr ':' suite elifs
else  ::= ϵ    | "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 ::= ϵ | "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();
```

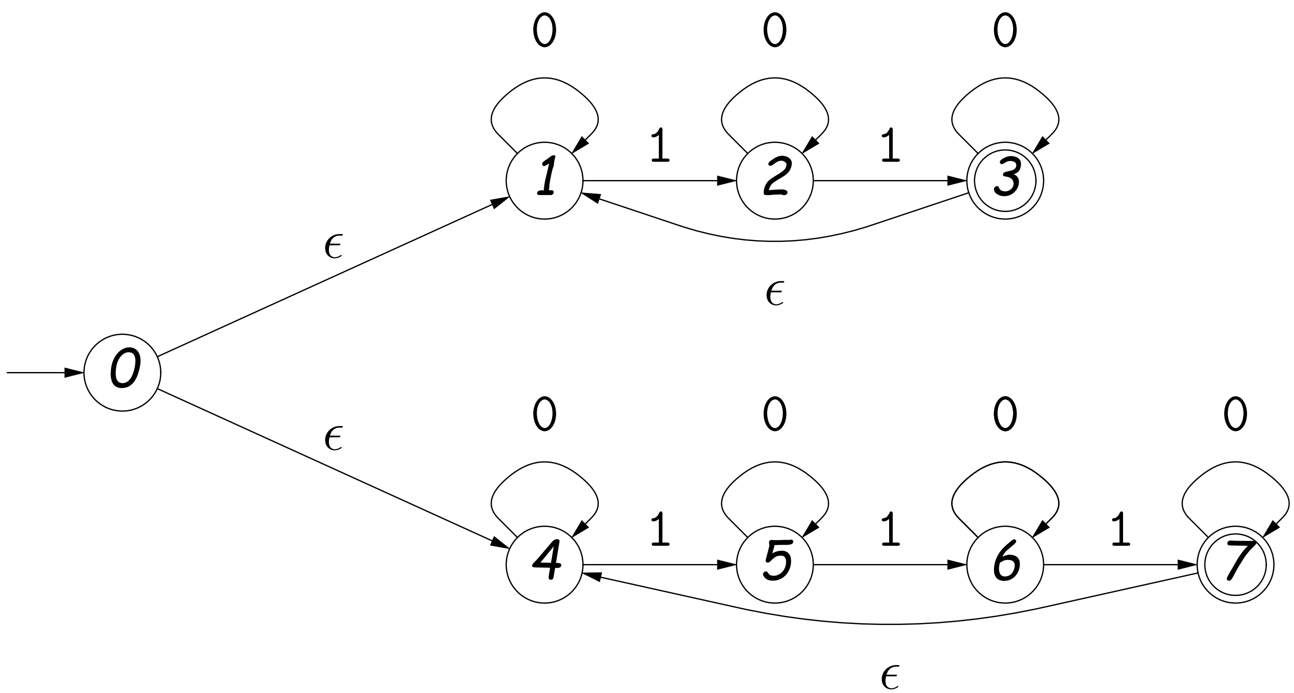
as

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

But it's not exactly clear, is it?

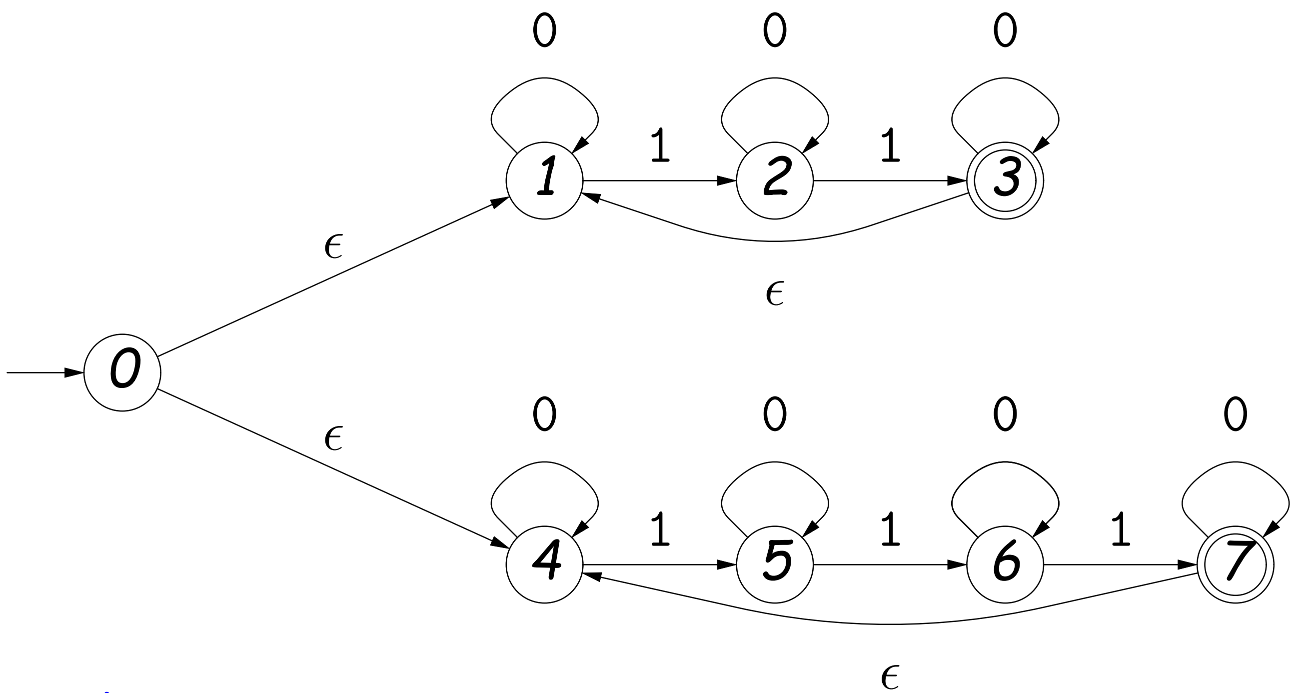
Puzzle: NFA to BNF

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?



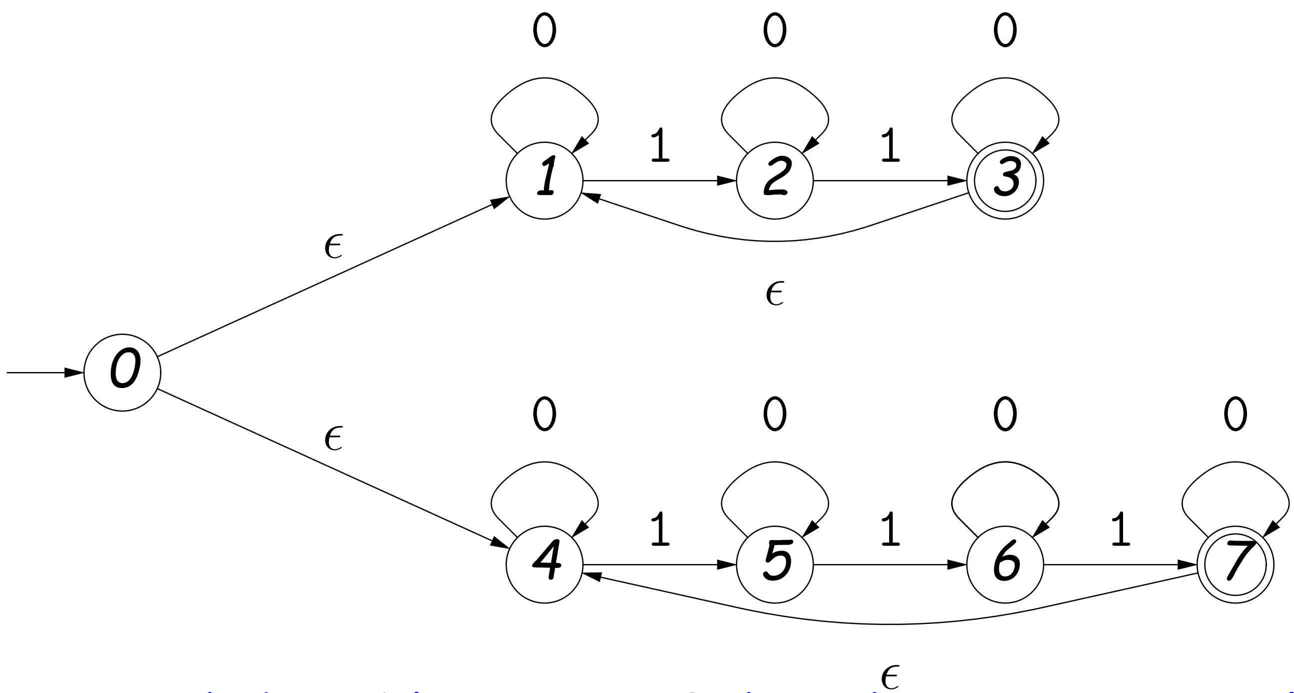
A conventional answer:

```
S ::=: S2s | S3s
Z ::=: '0' Z | ϵ
S2s ::= S2  | S2 S2s
S3s ::= S3  | 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:

```
S0 ::= S1 | S4                    S4 ::= '1' S5 | '0' S4
S1 ::= '1' S2 | '0' S1            S5 ::= '1' S6 | '0' S5
S2 ::= '1' S3 | '0' S2            S6 ::= '1' S7 | '0' S6
S3 ::= S1 | '0' S3 | ϵ            S7 ::= S4 | '0' S7 | ϵ
```

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