Languages and grammars: Why?

1. Lexical analysis can be defined by a grammar.
2. Syntax analysis (parsing) can be defined by a grammar.
3. No need to program these from scratch: just write the grammar.
4. But need to understand something about different grammars, etc.

Compiler generators for C and Java

	Programming Language	Lexical method	Parser method
Lex/Yacc	C	DFA	LALR(1)
Flex/Bison	C	DFA	LALR(1)
JavaCC	Java	DFA	LL(k)
SableCC	Java	DFA	LALR(1)
ANTLR	Java	LL(k)/LL(*)	LL(k)/LL(*)

LANGUAGES AND GRAMMARS

A grammar G has 4 parts: $G = (N, T, P, S)$

- N: Set of nonterminal symbols
- T: Set of terminal symbols
- P: Set of productions
- S: Start symbol ($S \in N$)

Nonterminals and terminals are disjoint: $N \cap T = \emptyset$

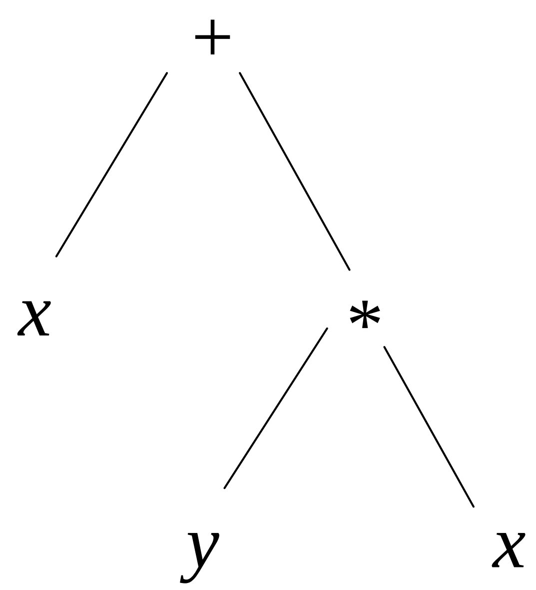
Production P: $(N \cup T)^+ \to (N \cup T)^*$

Language $L(G) = \{ w \in T^* \mid S \Rightarrow_G w \}$

Example: arithmetic expressions

$$G = (\ \{E, M, F\}, \{x, y, +, *, (,)\}, P, E\ )$$

$$
\begin{aligned}
P = \{\ & E \rightarrow M, \\
& E \rightarrow E + M, \\
& M \rightarrow F, \\
& M \rightarrow M * F, \\
& F \rightarrow x, \\
& F \rightarrow y, \\
& F \rightarrow (\ E\ )\ \}
\end{aligned}
$$



$$x + y * x \ \in \ L(G)$$

Parse trees (Syntax trees)

The full parse tree for the string $x + y * x$ is:

$E \to M$

$E \to E + M$

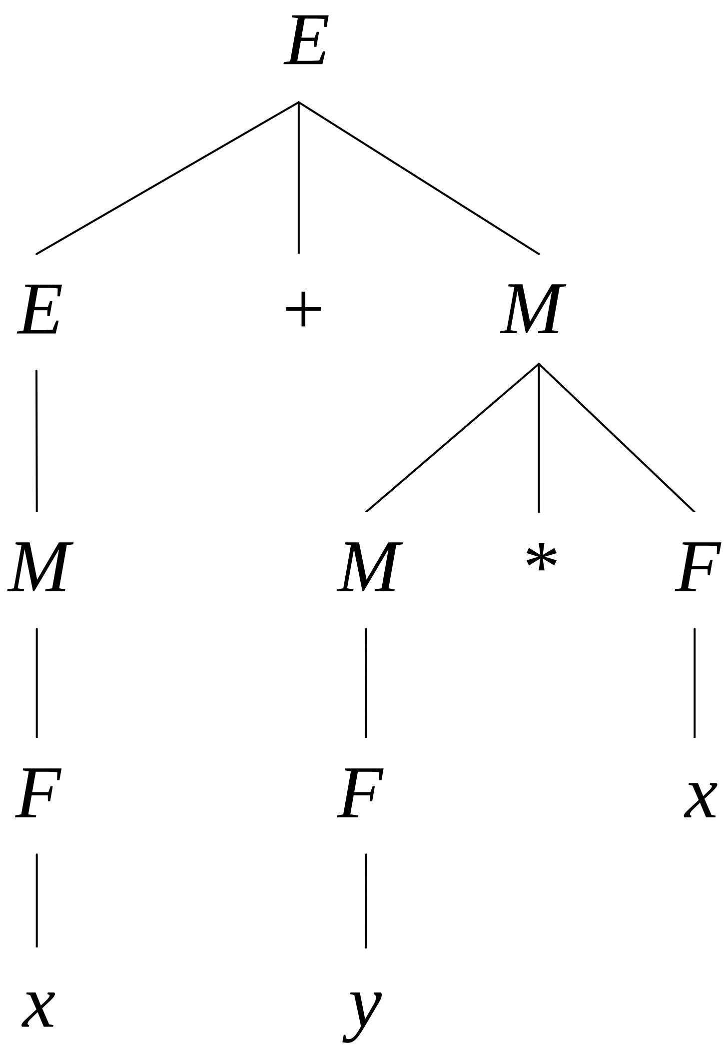
$M \to F$

$M \to M * F$

$F \to x$

$F \to y$

$F \to (E)$



The Chomsky hierarchy

- **Type 0: recursively enumerable**

  $$\alpha \rightarrow \gamma$$

- **Type 1: context-sensitive**

  $$\alpha A \beta \rightarrow \alpha \gamma \beta$$

- **Type 2: context-free**

  $$A \rightarrow \gamma$$

- **Type 3: regular**

  $A \rightarrow a,\ A \rightarrow Ba,\ A \rightarrow \varepsilon$ (No recursion)

Where $\alpha, \beta, \gamma \in (N \cup T)^*$ and $A, B \in N$ and $a \in T$.

Recognizers

- **Type 0-1**

  Not used for artificial languages.

- **Type 2: context-free**

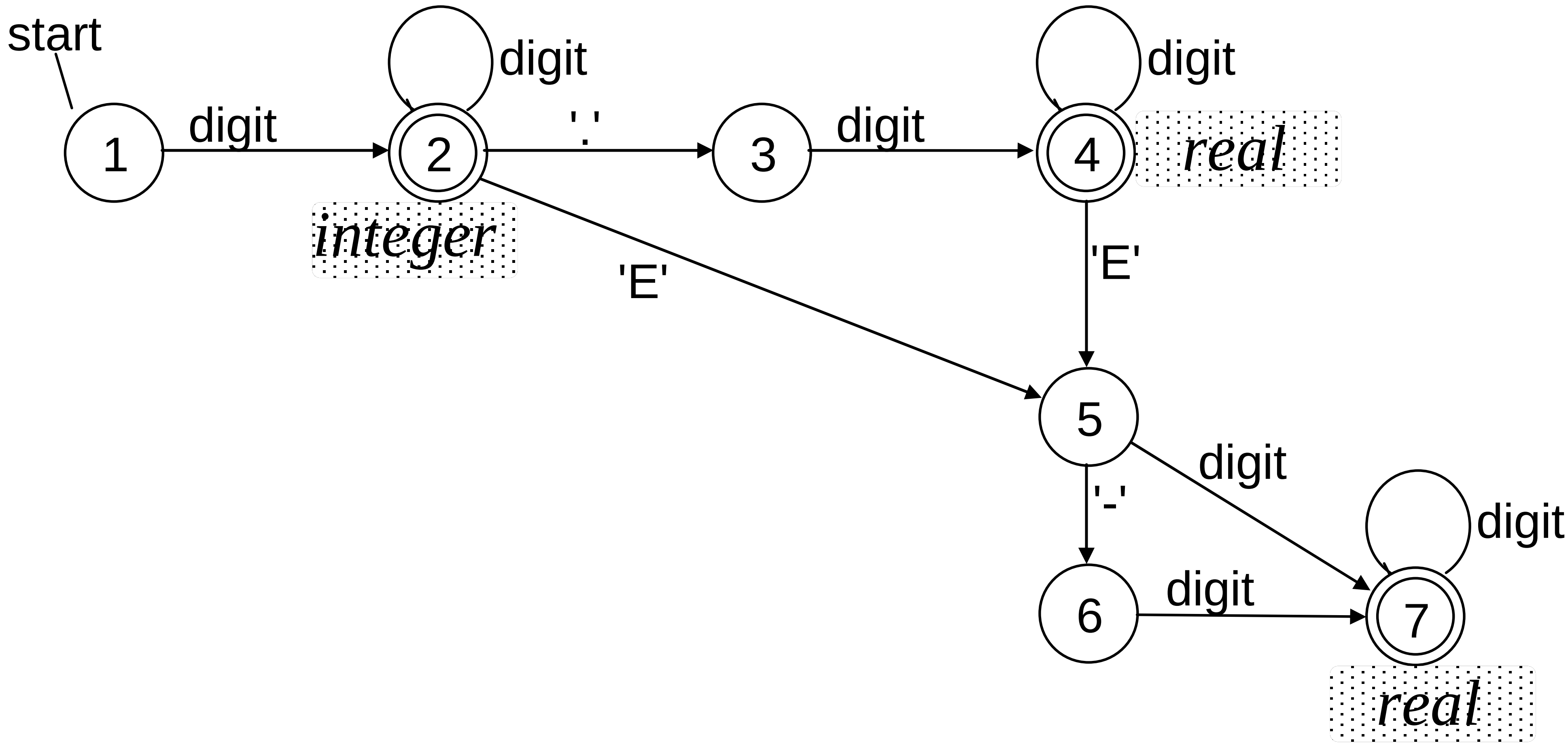
  Can be recognized by a *nondeterministic pushdown automaton*.

- **Type 3: regular**

  Can be recognized by a *finite automaton*.

  See COMS11700.

Deterministic finite automaton (DFA) to recognize integer and real numbers:



Backus Naur Form (BNF)

BNF is a notation to describe context-free grammars:

- Nonterminal symbols enclosed in <brackets>
- Terminal symbols may be in 'quotes' (and are not defined)
- `::=` separates nonterminal from its definition
- `|` separates alternatives

BNF Example

$E \rightarrow M$

$E \rightarrow E + M$

$M \rightarrow F$

$M \rightarrow M * F$

$F \rightarrow x$

$F \rightarrow y$

$F \rightarrow (E)$

```
<e> ::= <m> | <e> '+' <m>

<m> ::= <f> | <m> '*' <f>

<f> ::= 'x' | 'y' | '(' <e> ')'
```

Ambiguous grammars

A grammar is *ambiguous* if it can derive one sentence with more than one parse tree. E.g.:

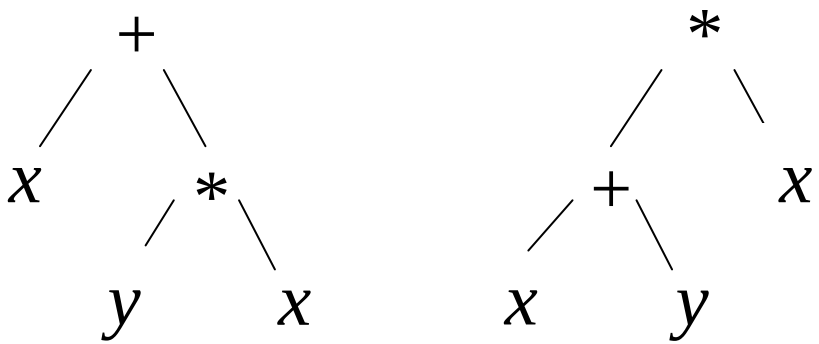
```
<letter> ::= 'a' | … | 'z'
<identifier> ::= <letter>+
<keyword> ::= 'i' 'f' | 'e' 'l' 's' 'e' | …
<token> ::= <keyword> | <identifier>
```

Two parses of “`if`”, “`else`”, etc.

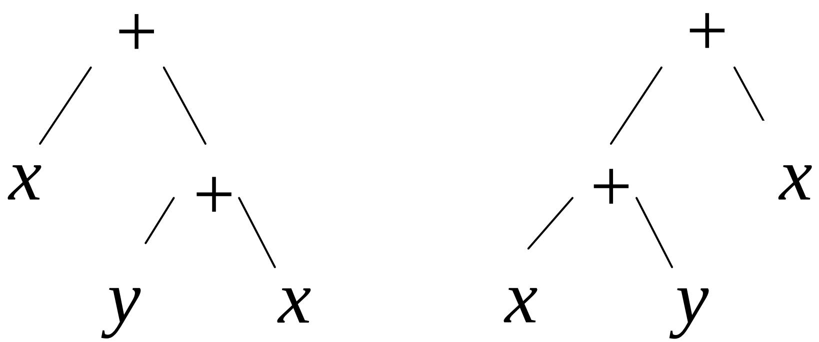
Another ambiguous grammar:

```
<exp> ::= <exp> '+' <exp> |
          <exp> '*' <exp> |
          'x' | 'y' | '(' <exp> ')'
```

Two parses of “x+y*x”:



Two parses of “x+y+x”:



Dealing with ambiguity

```
<exp> ::= <exp> '+' <exp> | <exp> '*' <exp> |
          'x' | 'y' |'(' <exp> ')'
```

- To give * higher **precedence** than +

```
<exp> ::= <term> | <exp> '+' <term>
<term> ::= <factor> | <term> '*' <factor>
<factor> ::= 'x' | 'y' | '(' <exp> ')'
```

- To make * and + right-**associative** (instead of left)

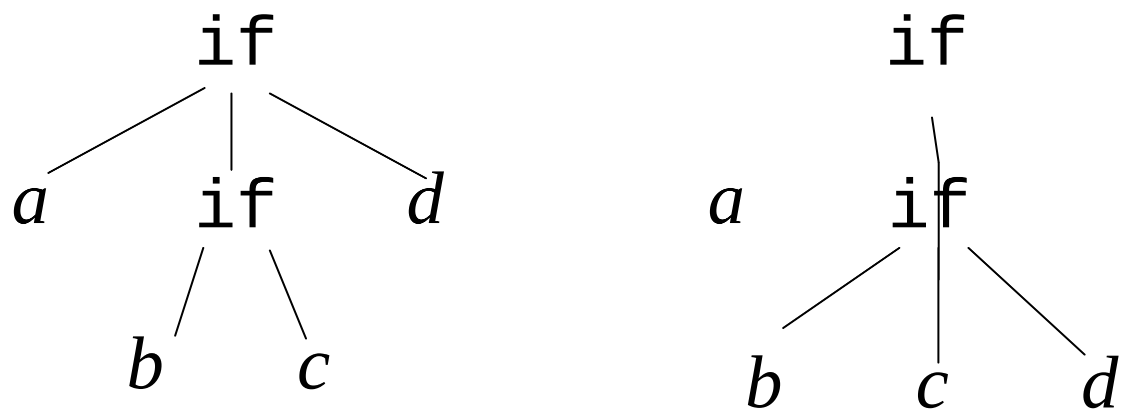
```
<exp> ::= <term> | <term> '+' <exp>
<term> ::= <factor> | <factor> '*' <term>
```

Another ambiguous grammar: the “dangling else”:

```
<st> ::= 'if' '(' <exp> ')' <st> ('else' <st>)?
```

Two parses of:

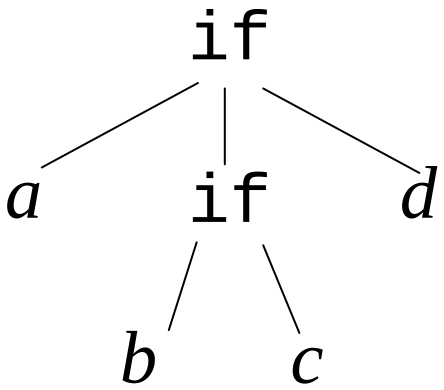
```
if (a) if (b) c; else d;
```

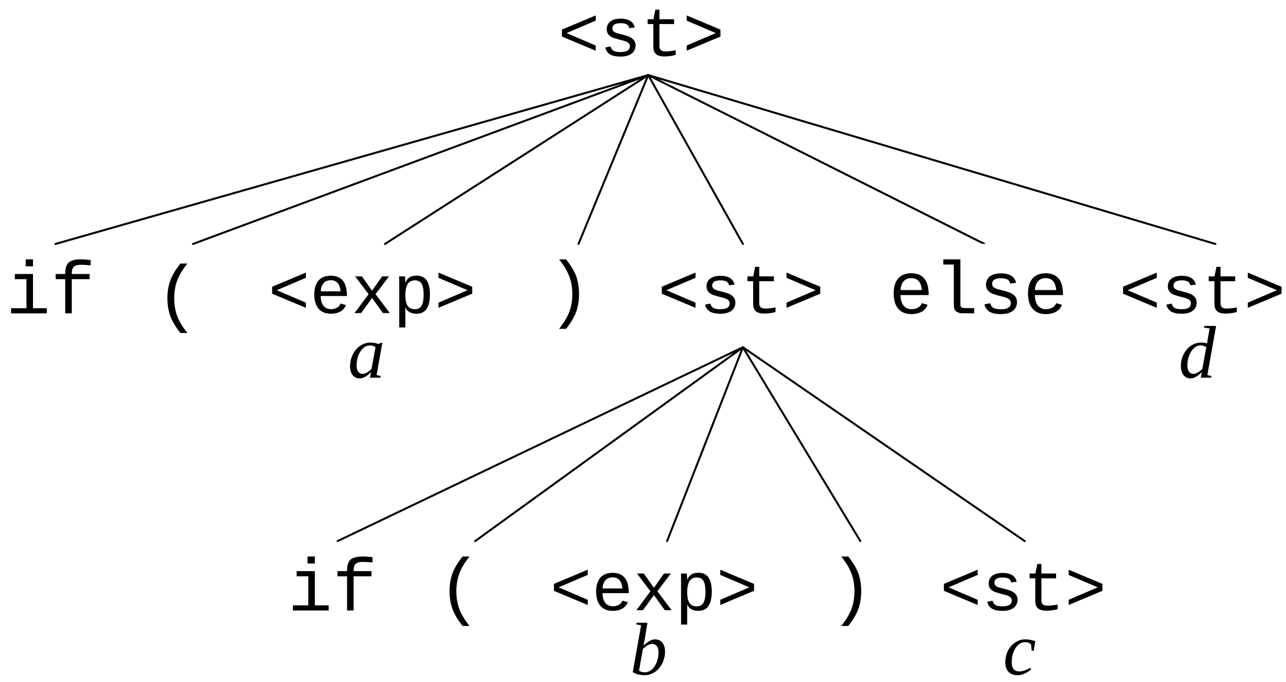


```
<st> ::= 'if' '(' <exp> ')' <st> ('else' <st>)?
```

```
if (a) if (b) c; else d;
```

Left: compact version. **Right: full parse tree.**





```
<st> ::= 'if' '(' <exp> ')' <st> ('else' <st>)?
```

- Java makes this unambiguous by transforming the grammar:

```
<st> ::= <simple> | <ift> | <ifte>
<sst> ::= <simple> | <iftes>
<ift> ::= 'if' '(' <exp> ')' <st>
<ifte> ::= 'if' '(' <exp> ')' <sst> 'else' <st>
<iftes> ::= 'if' '(' <exp> ')' <sst> 'else' <sst>
```

Extended BNF (EBNF)

Extended BNF also allows:

- *: zero or more times
- +: one or more times
- ?: zero or one times
- (): parentheses

$E \rightarrow M$	`<e> ::= <m> ('+' <m>)*`
$E \rightarrow E + M$	
$M \rightarrow F$	`<m> ::= <f> ('*' <f>)*`
$M \rightarrow M * F$	
$F \rightarrow x$	`<f> ::= 'x' | 'y' | '(' <e> ')'`
$F \rightarrow y$	
$F \rightarrow (E)$	

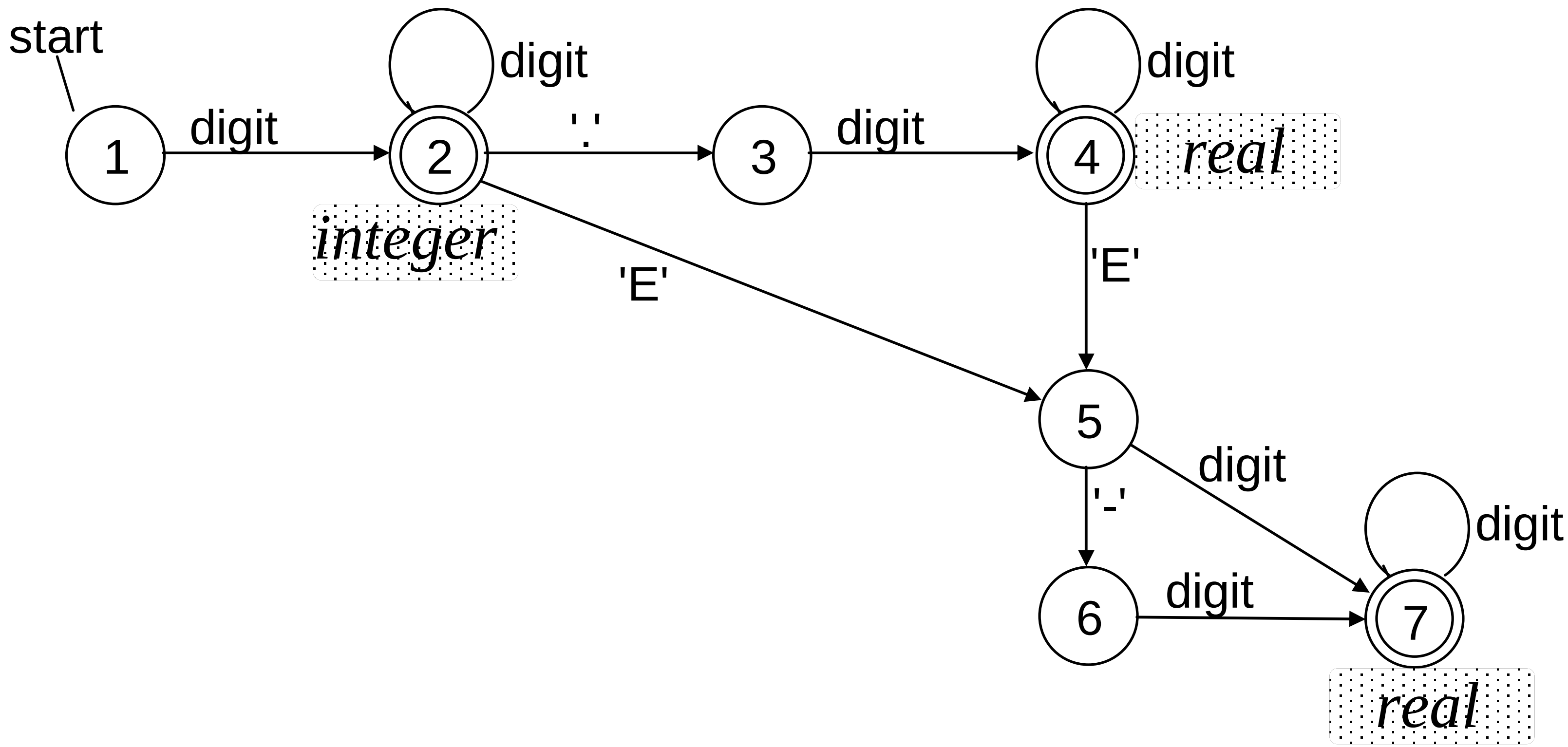
Example: regular grammar for (integer and real) numbers

```
<integer> → <digit>+

<real> → <integer> '.' <integer> <exponent>?
        | <integer> <exponent>

<exponent> → 'E' '-'? <integer>
```

Can be translated to this DFA:



Transforming EBNF to BNF

- Parentheses:
  `…(v)…` $\Rightarrow$ `…vs…` where `vs ::= v`
- Repetition: * is same as optional +:
  `…v*…` $\Rightarrow$ `…v+?…`
- Repetition can be defined by recursive rule:
  `…v+…` $\Rightarrow$ `…vs…` where `vs ::= v | v vs`
- For an optional item, we can define two alternatives:
  `x ::= u v? w` $\Rightarrow$ `x ::= u w | u v w`

This generates 2^n alternatives if original contains *n* optional items.

Test

```
<e> ::= <e> '*' 'id'
<e> ::= 'id'
<e> ::= '*' 'id'
<e> ::= '(' <e> ')'
<e> ::= '(' 'id' ')' <e>
```

1. **Is this grammar ambiguous?**
2. **Draw the parse tree(s) for the string**

   `(list)*list`

   where `list` **is a token with lexical type** `'id'`

Answer