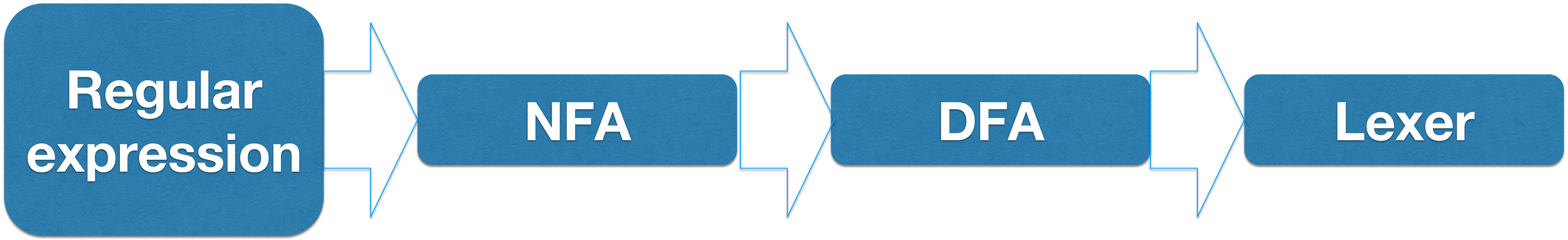
Regex engines


Regular expression
NFA
DFA
Lexer

code for DFA

- Using a transition table, it is straightforward to write a program to recognize strings in a regular language

State	Character		
	a	b	c
1	2		
2		3	
3			4
4	2		4

```
state = initial_state; //start state of FA
while (true) {
   next_char = getc();
   if (next_char == EOF) break;
   next_state = T[state][next_char];
   if (next_state == ERROR) break;
   state = next_state;
}
if (is_final_state(state))
   //recognized a valid string
else
   handle_error(next_char);
```

lookahead

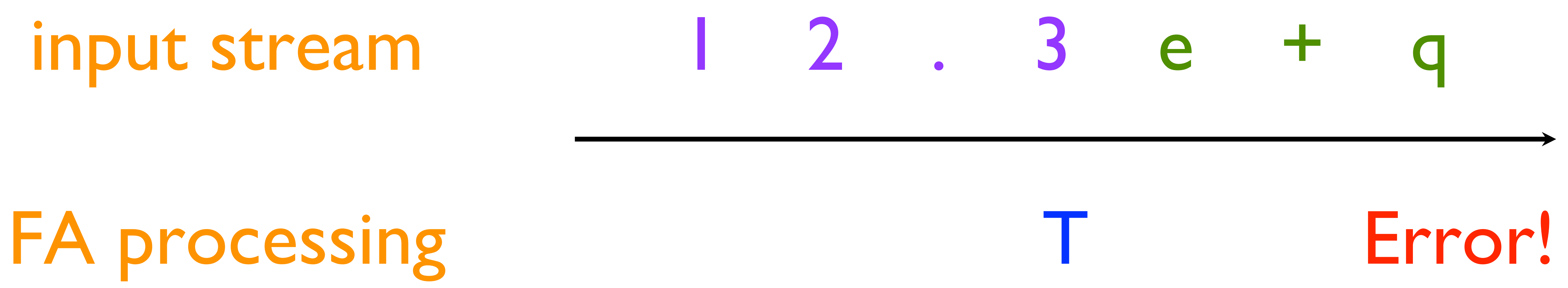
- Up until now, we have only considered matching an entire string to see if it is in a regular language
  - What if we want to match multiple tokens from a file?
    - Multiple token definitions
  - Distinguish between int a and inta
- We need to *look ahead* to see if the next character belongs to the current token
  - If it does, we can continue
  - If it doesn’t, the next character becomes part of the next token

breaking ties

- What if we can add the next character to the current token *or* end the current token?
- Scanner engine has tie breaking rules
    - Always make a token as long as possible (or as short as possible—this is what Python's regex engine does)
    - If multiple possible tokens match, give them a priority order (e.g., prioritize tokens defined first)

general approach

- Remember states (T) that can be final states
- Buffer the characters from then on
- If stuck in a non-final state, back up to T, restore buffered characters to stream
- Example: 12.3e+q



antlr

- A tool for building scanners and parsers
  - Language for defining tokens, automatically converted into Java, C, Python, etc.
  - An example of compiling one high level language to another!
- Tokens
  - Token definition: tokenName : regex1 | regex2 | …
  - Define tokens in precedence order
- Character classes
  - Look similar to token definitions
  - **fragment** characterClassName : regex1 | regex2 ...
  - Can use character classes when defining tokens

parsing

- We've covered how to tokenize an input program
- But how do we decide what tokens actually say?
- How do we recognize that

```
IF ID(a) OP(<) ID(b) { ID(a) ASSIGN LIT(5) ; }
```

  is an if-statement?

- We need something more powerful!