

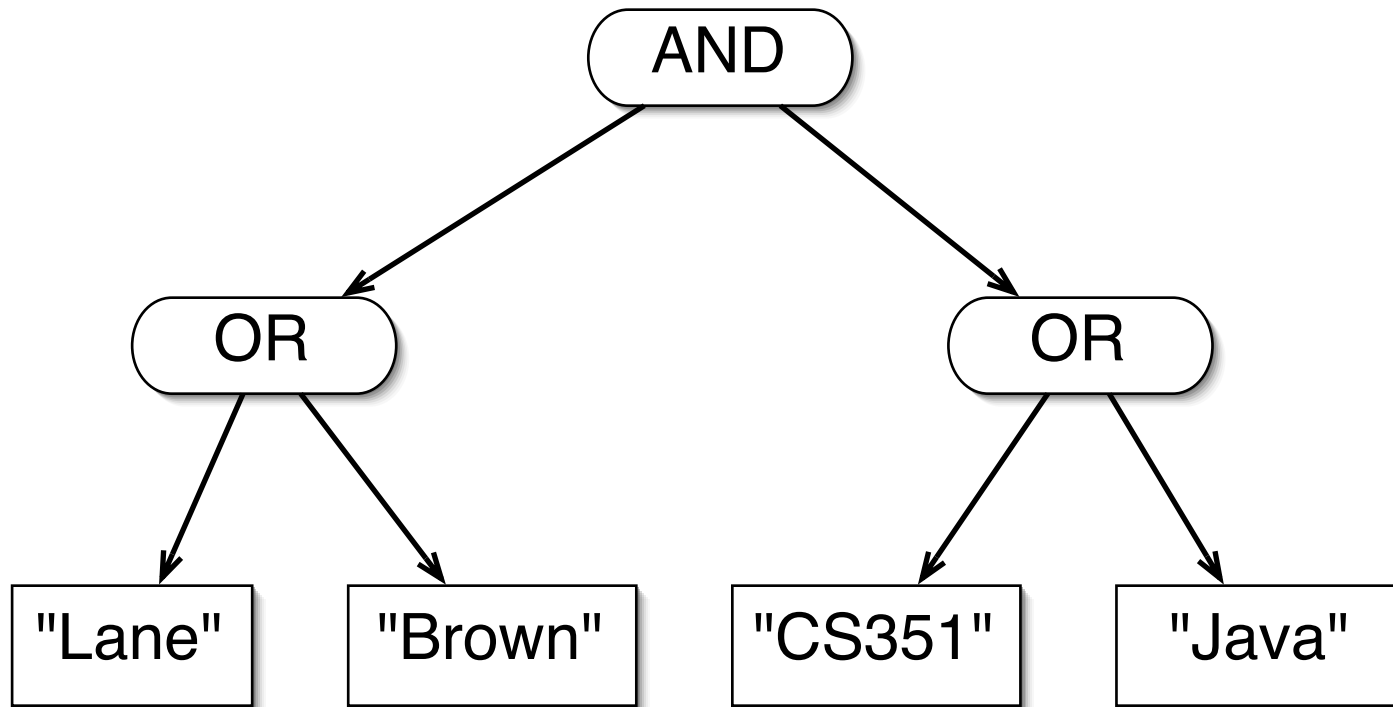
Lexing & Parsing (the intro version)

What you *have*

- Input is string (via `JTextField.getText()`)
 - “(Lane OR Brown) AND (CS351 OR Java)”
- Useful to transform it to a Reader (`StringReader`)
 - Gives uniform interface; isolates parser from source of data -- could read from file or network as easily as from string
- Char stream gives you data one character at a time
 - `char Reader.read()`

What you want

- Parse tree
- Data structure that reflects syntactic structure of input phrase

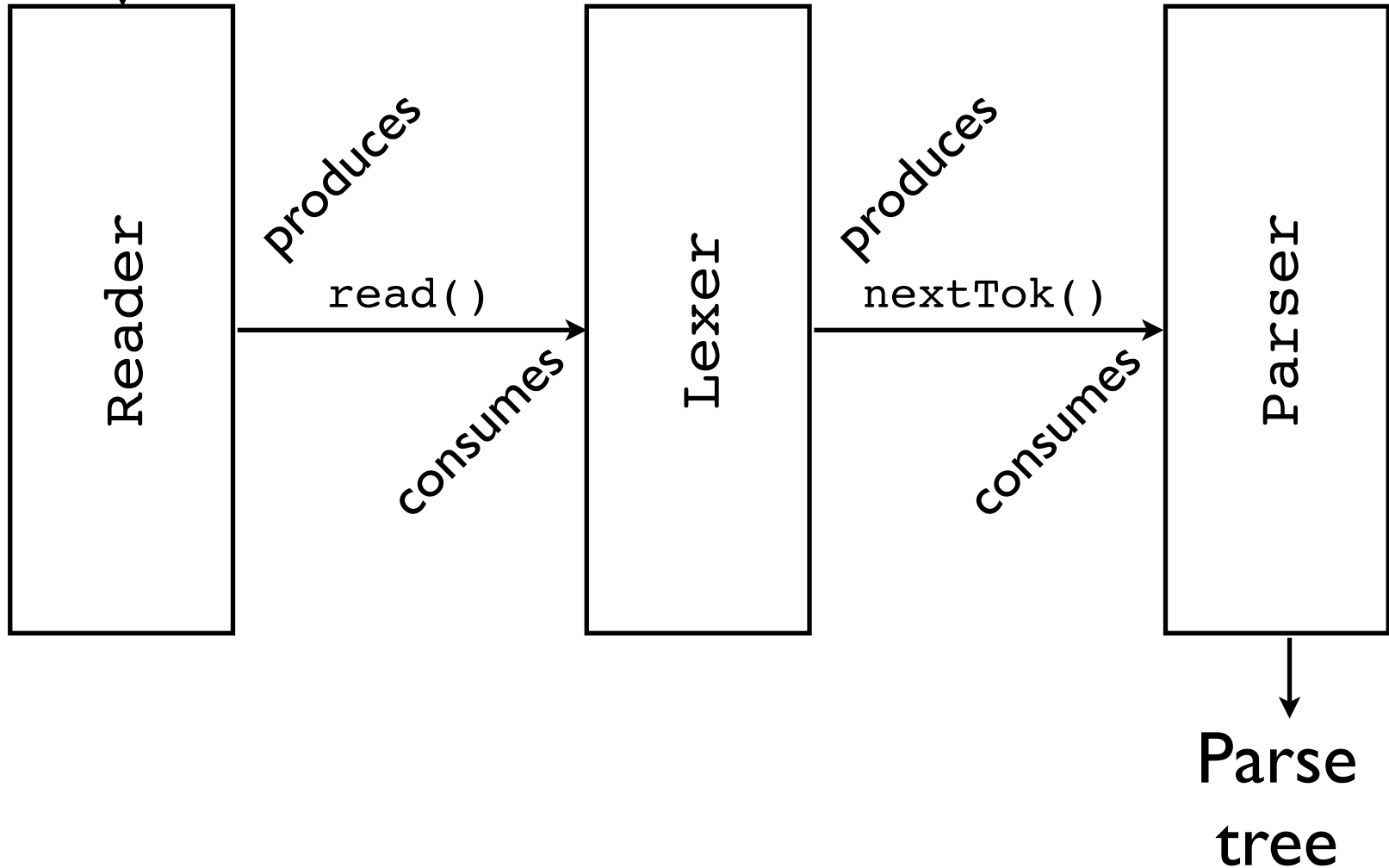


2 Questions

- How do you get from a character stream (Reader) to a parse tree?
- Why do you want a parse tree anyway?

The parsing pipeline

Character
data (String)



Roles of the modules

- **Reader:**
 - Provides a uniform interface to *character* data
 - Generates one character at a time, on request
 - Notifies caller when stream empty (EOF)
- **Lexer** (lexical analyzer):
 - Converts groups of characters into *tokens*
 - Generates one token at a time, on request
 - Notifies caller when no more tokens are avail.
- **Parser:**
 - Converts groups of tokens into *parse trees*
 - Generates a complete parse tree, on request

In call-stack notation

Parser

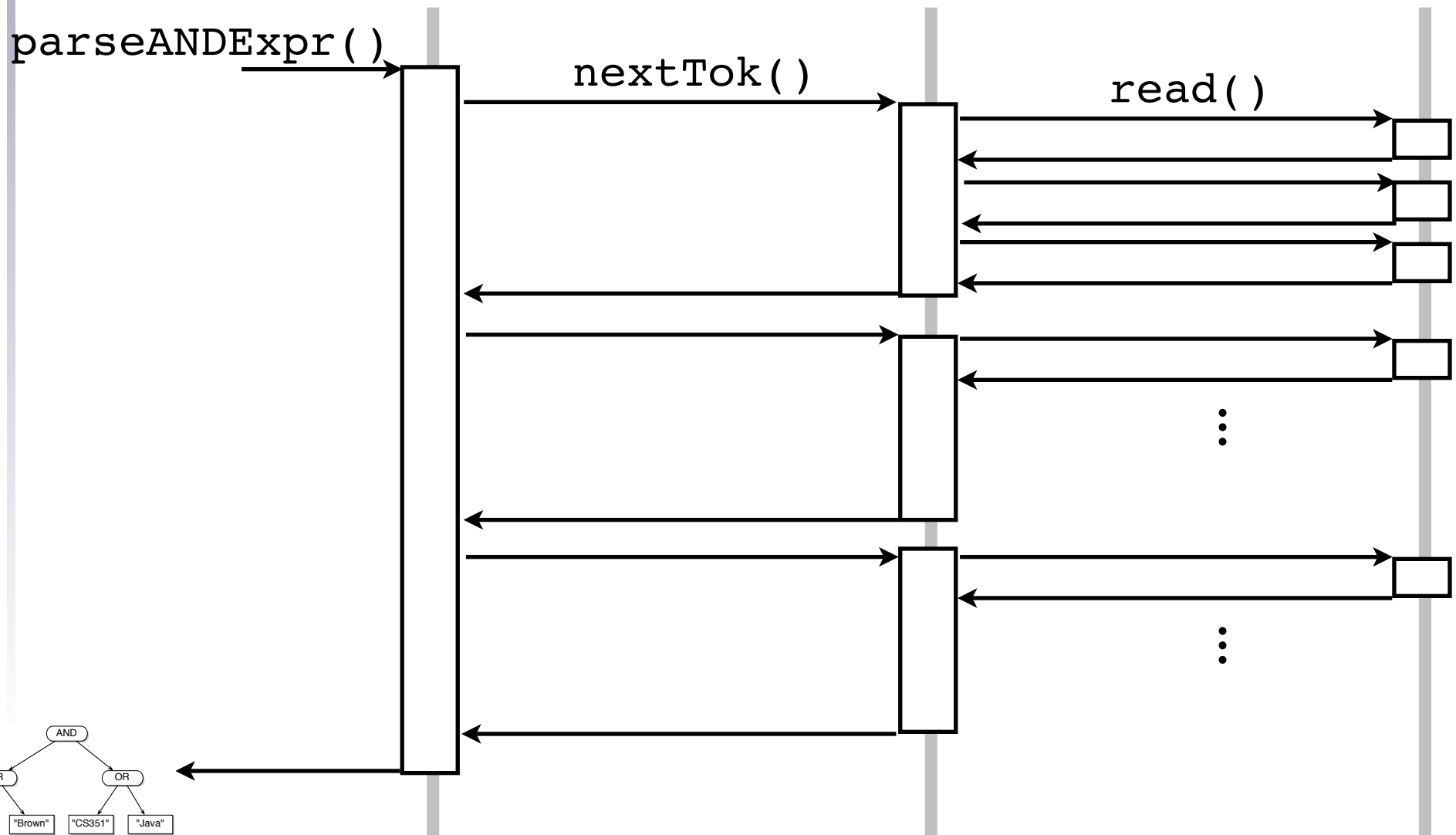
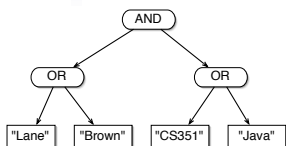
Lexer

Reader

`parseANDExpr()`

`nextTok()`

`read()`



Why Lexer and Parser?

- What's the difference between the lexer and the parser?
- View 1 (abstract): Not much -- both are stream transforms
- View 2 (practical): A lot.
 - Tokens and parse trees are conceptually different beasts
 - Can implement lexer w/ DFA (finite state machine)
 - Need recursion to handle full parsing (CFG; stack machine; recursive descent parsing)

The joy of lex

- Job of lexer (a.k.a., tokenizer) : convert stream of characters to stream of tokens
- Tokens are (roughly) things that can be described with regular expressions (regexps)
 - "haruspex"
 - "[a-zA-Z]+"
 - "[0-9]+(.[0-9]+)?"
- Key idea: state machine determines what to do next
 - Given [state, char], know:
 - Whether to complete and return token
 - What next state should be

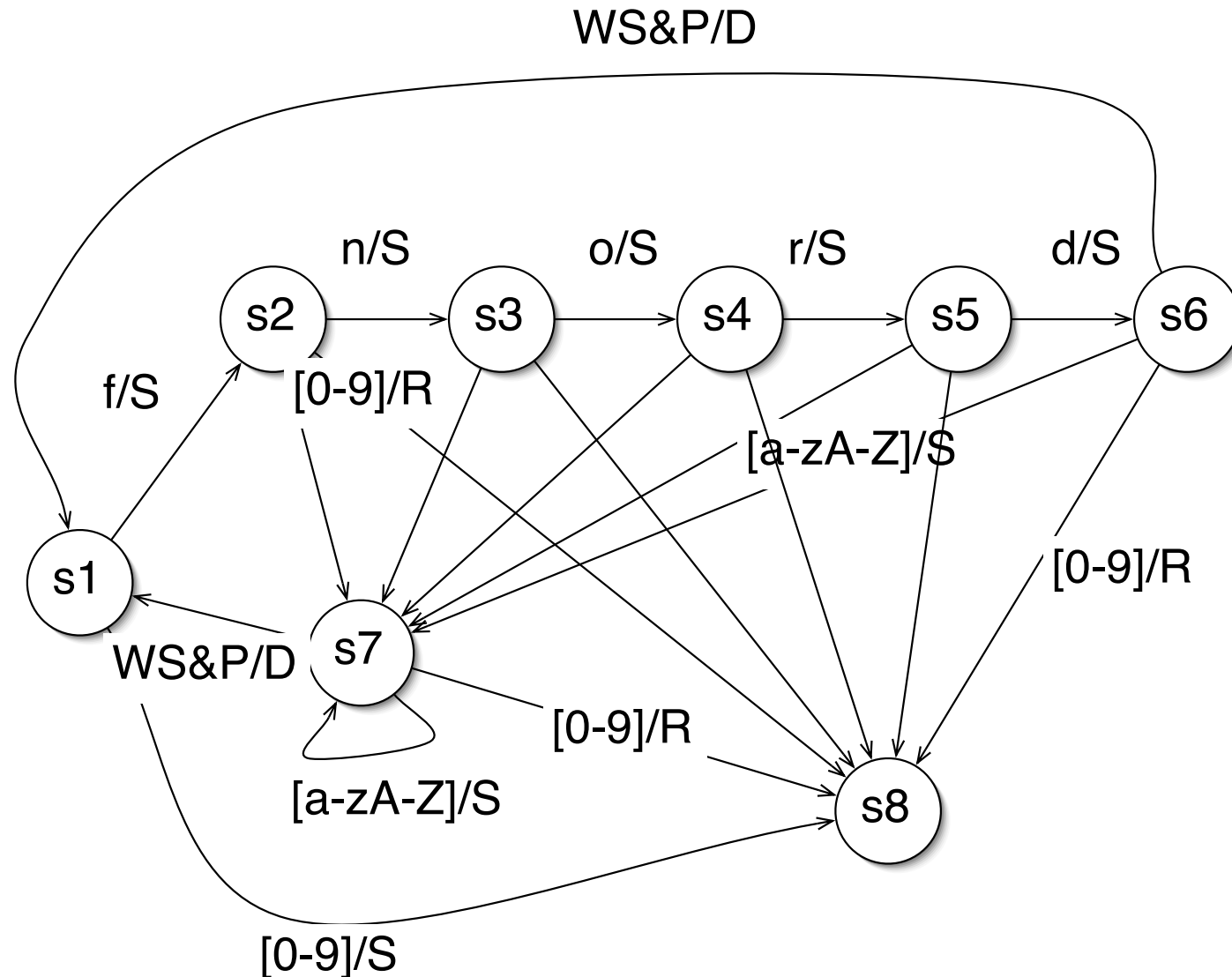
A simple token set

- Suppose you want to lex the following token types:
 - `TT_FNORD := "fnord"`
 - `TT_WORD := "[a-zA-Z]*"`
 - `TT_NUMBER := "[0-9]+"`
 - Any other char separates tokens, but is otherwise discarded
- Examples:
 - `"123kumquat 9" => "123" "kumquat" "9"`
 - `"fnordling" => "fnordling"`
 - `"fnordling" => "fnord" "1" "ing"`

Definitions

- **Shift:** Save current character on accumulator (`StringBuffer`) and continue
- **Return:**
 - Create new token from accumulator
 - Clear accumulator
 - Shift current character onto accumulator
 - Return new token
- **DRet (drop and return):**
 - Create new token from accumulator
 - Clear accumulator
 - Return new token

The state machine



Plus a *lot* more -- very complex diagram. Lot of cases to consider..

Turning it into code

```
while (1) {
    char c=inputStream.read();
    switch (currState) {
        case ST_S1:
            if (c=='f') {
                currState=ST_S2;
                doShift(c);
                next;
            }
            if (Character.isDigit(c)) {
                currState=ST_S8;
                doShift(c);
                next;
            }
            // etc.
```

Turning it into code

```
case ST_S6:
    if (Character.isDigit(c)) {
        currState=ST_S8;
        Token t=doReturn(c,TT_FNORD);
        return t;
    }
    if (Character.isLetter(c)) {
        currState=ST_S7;
        next;
    }
    // etc.
case ST_S7:
    if (Character.isDigit(c)) {
        currState=ST_S8;
        Token t=doReturn(c,TT_WORD);
        return t;
    }
```