Choices for Parsing Algorithms

- Grammar: E → (E + E) | **num**
- Sentence: (2 + 3)
- We want a parsing algorithm that makes a single left-to-right scan of the sentence and runs in time *linear* in the number of tokens in the sentence.

- Top-down parser
  - *Grow* a parse tree top-down beginning with the start symbol (the root) and ending with the given sentence (the leaves).
  - Naturally traces a leftmost derivation.
- Bottom-up parser
  - *Contract* a parse tree bottom-up beginning with the given sentence and ending with the start symbol.
  - Naturally traces a rightmost derivation (in reverse).

Choosing A Production To Use

- Grammar: E → (E + E) | **num**
- Sentence: (2 + 3)
- To reach the leftmost derivation, how do we decide which production to use in the first step?

  E → (E + E)

  E → **num**
- Answer: Examine the next unread token in the input stream. Three cases:

num:	Use the production E → **num**.
'(':	Use the production E → (E + E).
Otherwise:	Parsing error.

- This rule works for *all* derivation steps, not just the first.
- This next unread token in the input stream is called the "look-ahead". In general, we could look ahead k tokens, with $k \geq 1$.

LL(1) Grammar

- A grammar that allows a deterministic top-down parser that scans the sentence **L**eft-to-right and produces a **L**eftmost derivation using a look-ahead of one (**1**) token is called an **LL**(**1**) grammar.

- Two views of parser for LL(1) grammar
  - *Sentence* view: Determine a leftmost derivation of the input, reading the tokens from left to right while looking ahead at most one input token.
  - *Tree* view: Beginning with the start symbol, grow a parse tree top-down in left-to-right preorder while looking ahead at most one input token beyond the input prefix matched by the parse tree derived so far.

- Such a deterministic top-down parser for an LL(1) grammar is called a recursive-descent parser (or a predictive parser).

Recursive-Descent Recognizer

E → (E + E) | **num**

```
token = input.read(); //global variable
parse_E();

// precondition: global variable "token" has look-ahead token
boolean parse_E() {
   switch (token) {
        case num: token = input.read(); return true;
        case '(':
                token = input.read();
                parse_E();
                if (token != '+') throw new ParseError();
                token = input.read();
                parse_E();
                if (token != ')') throw new ParseError();
                token = input.read();
                return true;
        default: throw new ParseError();
     }
}
// postcondition: global variable "token" has look-ahead token
```

Non-LL(1) Grammar

- Consider the grammar
  - $S \rightarrow E + S \mid E$
  - $E \rightarrow$ **num** $\mid (S)$
- Consider the two derivations
  - $S \Longrightarrow E \Longrightarrow (S) \Longrightarrow (E) \Longrightarrow (3)$
  - $S \Longrightarrow E + S \Longrightarrow (S) + S \Longrightarrow (3) + E \Longrightarrow (3) + 4$
- How can we decide between the productions

  $S \rightarrow E$ and $S \rightarrow E + S$

  in the first derivation step based on one (or even some finite number k) of look-ahead tokens?
- We can't!
  - This grammar is not LL(1).
  - In fact, this grammar is not LL(k) for any k.
  - But it can be converted into an equivalent LL(1) grammar.

Making a Grammar LL(1)

S → E+S
S → E
E → **num**
E → (S)

S → ES’
S’ → ε
S’ → +S
E → **num**
E → (S)

- Two grammars G_1 and G_2 are said to be equivalent if $L(G_1) = L(G_2)$.

- Left-factoring
  - Factor the common prefix E of S.
  - Add a new non-terminal S’ for what follows that prefix.

- Convert left-recursion to right-recursion.

- *Caveat*: Not all context-free languages have an LL(1) grammar.