General Organization of Top-Down Parser

- Input: LL(1) grammar $G = (N, T, P, S)$.

- The global variable *token* contains the look-ahead token.
- One method for each non-terminal $n \in N$.
  - Pre-condition: Variable *token* has look-ahead token.
  - Action: Consume a sequence of terminals that can be derived from n.
  - Post-condition: Variable *token* has look-ahead token.
  - The methods are mutually recursive.
- The method body is a big `switch` statement.
  Each `case` of the `switch`:
  - Handles one possible look-ahead token (say, $t \in T$).
  - Invokes parsing actions for some production $p \in P$ of the form $n \to \alpha$.
- Question: How do we determine which production to use for the combination (n, t)?

Abstraction: Predictive Parsing Table

S → ES'
S' → ε
S' → +S
E → num
E → (S)

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ **num**		→ (S)		

- One row for each non-terminal $n \in N$.
- One column for each symbol in $T \cup \{\$\}$.
- Use the production in Table$[r, c]$ when expanding non-terminal r with look-ahead token c.
- Empty table entries are invalid: throw a parsing error.

- Given the parsing table, it is easy to generate the recursive-descent parser.

Recursive-Descent Parser (1 of 3)

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

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

lookahead token

```
void parse_S() {
   switch (token) {
      case num: case ‘(’:
            parse_E();
            parse_S’();
            return;
      default: throw new ParseError();
   }
}
```

Recursive-Descent Parser (2 of 3)

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

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

```
void parse_S’() {
   switch (token) {
      case ‘+’:
            token = input.read();
            parse_S();
            return;
      case ‘(’: case $: return;
      default: throw new ParseError();
   }
}
```

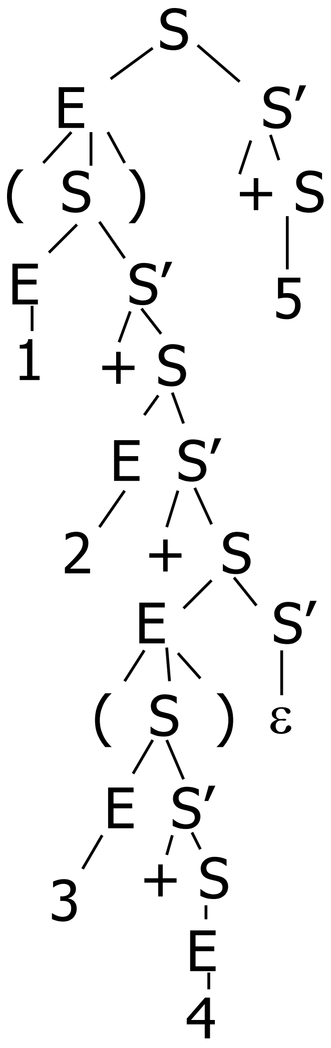
Recursive-Descent Parser (2 of 3)

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

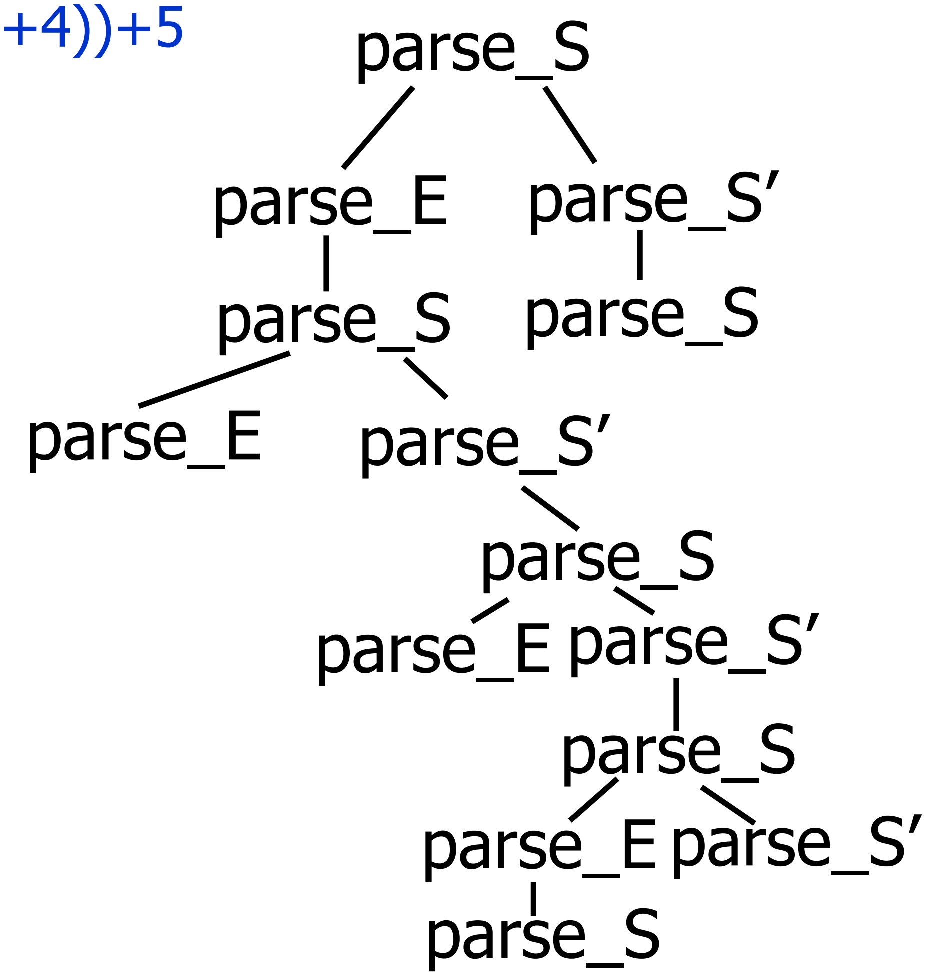
	num	+	(	)	$
S	→ ES’		→ ES’		
S’		→ +S		→ ε	→ ε
E	→ **num**		→ (S)		

```
void parse_E() {
   switch (token) {
      case num: token = input.read(); return;
      case ‘(’:
        token = input.read(); parse_S();
        if (token != ‘)’) throw new ParseError();
        token = input.read(); return;
      default: throw new ParseError();
   }
}
```

The Parse Tree Is Just The Call Tree

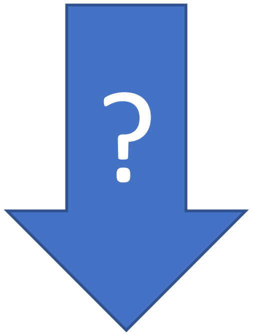


(1+2+(3+4))+5



Constructing Parsing Tables

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



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