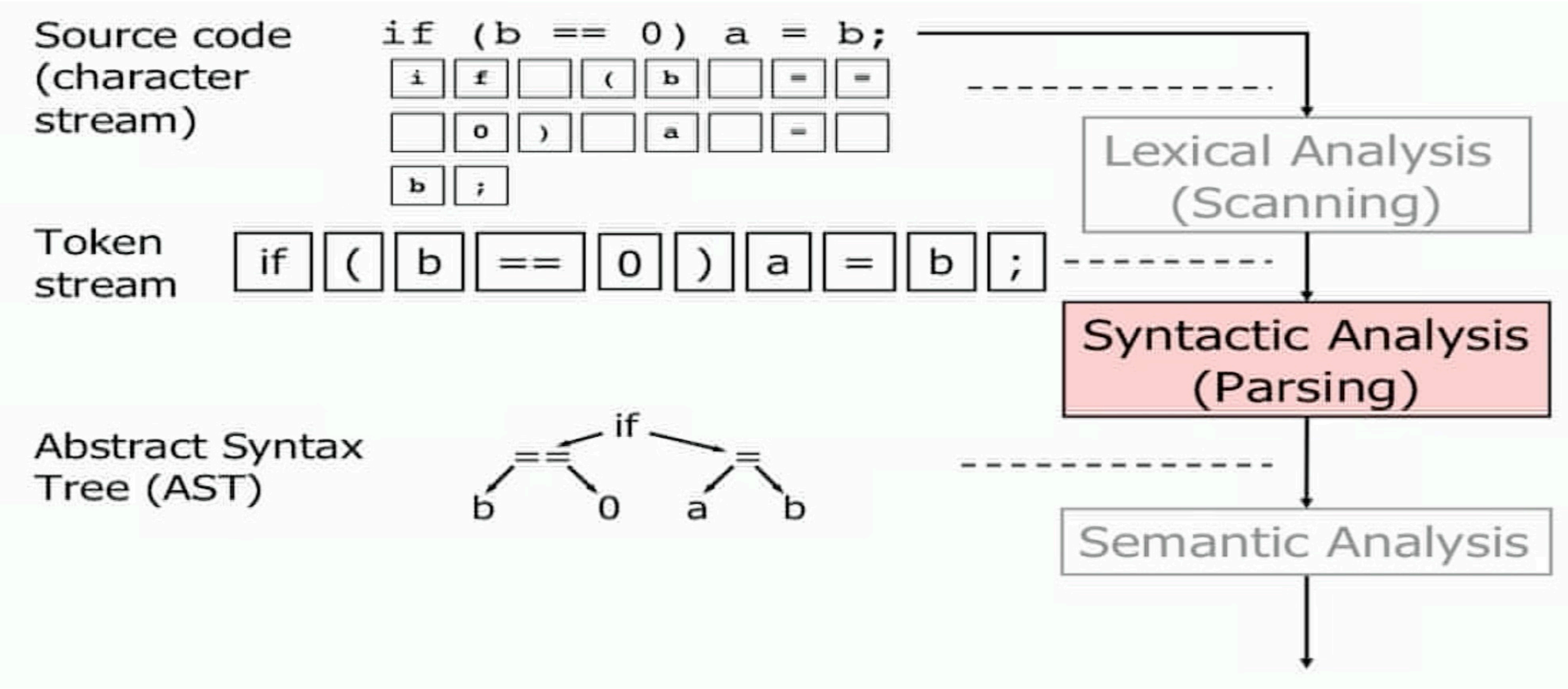
PRACTICAL PARSING

Motivation question:

How does production code turns into Abstract Syntax Tree (AST) ?


Source code (character stream)
if (b == 0) a = b;
i f (b = = 0) a = b ;
Token stream
if (b == 0) a = b ;
Lexical Analysis (Scanning)
Syntactic Analysis (Parsing)
Abstract Syntax Tree (AST)
if
==
=
b
0
a
b
Semantic Analysis

How does this connect to previous lesson (LiveOak grammar) ?

Recall from previous lesson:

- Grammar (G) is finite set of instructions
- Language (L) is infinite set of strings, characters

⇒ L(G) is notion for a programming language.

Now, we are talking about Syntatic Analysis (Parsing), which has 2 steps:

a) Recognition:

checks if a sentence (a "tokenized" line of production code) conforms to grammar rules.

Formally: $s \in L(G)$?

tokenized sentence

b) Parsing:

constructs the AST, proofs that $s \in L(G)$

Example:

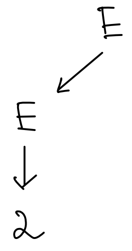
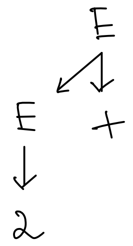
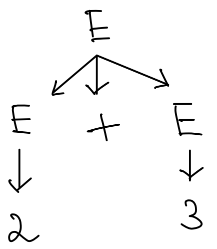
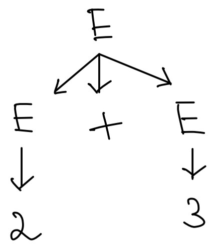
Consider this simple grammar: $E \rightarrow (E + E) \mid num$

and this sentence: $(2 + 3)$

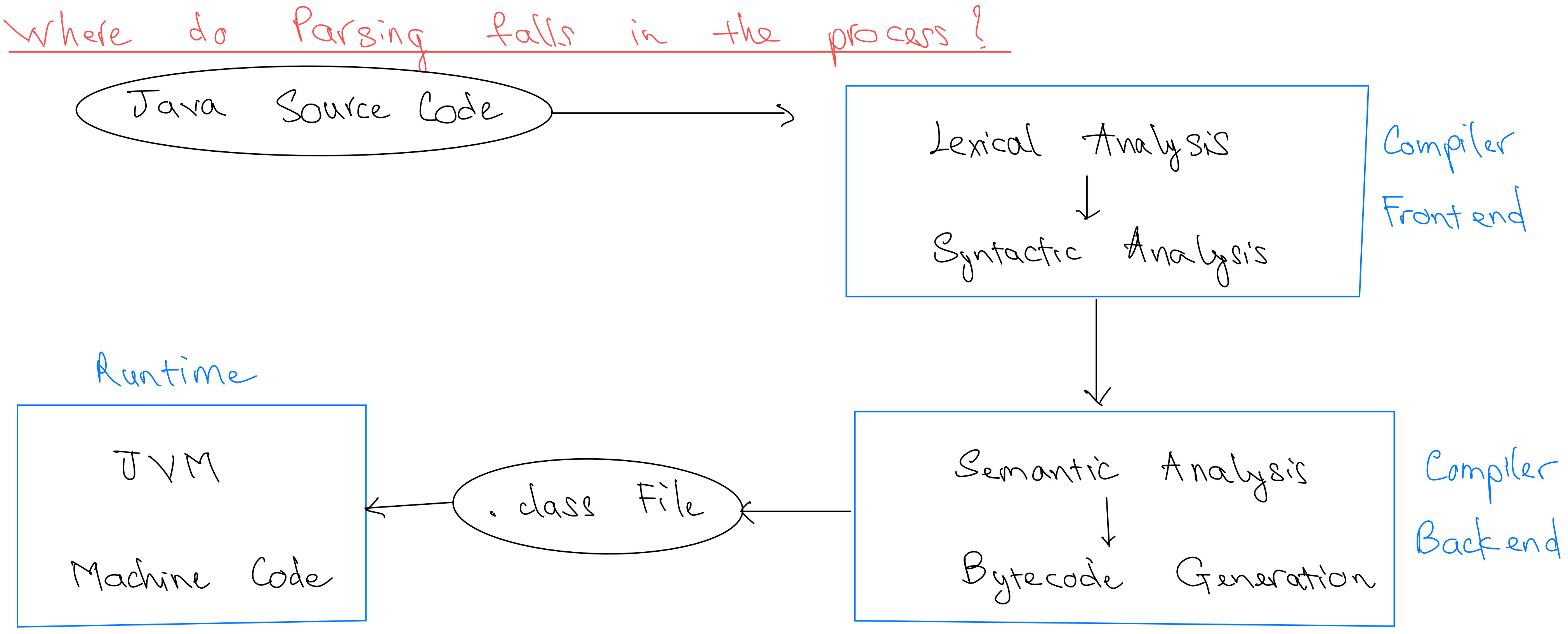
Here is how Syntactic Analysis works:

- Assuming Lexical Analysis already tokenized the sentence
- Recognition happens as we feed token by token to the Parser
- The AST is built incrementally during this process

Specifically, here is how Parsing works:

Step	Token	Stack	Action	AST	Reason
1	(	(	Shift	E	Opening parenthesis, expect more
2	2	(2	Shift		Shift 2 onto stack
		(E	Reduce		Matches 'num', so reduce to E
3	+	(E +	Shift		shift "+" onto stack, doesn't complete a rule yet
4	3	(E + 3	Shift		Shift 3 onto stack
		(E + E	Reduce		Matches 'num', so reduce to E
5	)	(E + E)	Shift		Shift "(" onto stack
		E	Reduce		Matches "(E + E)", so reduce to E
6	[end]	E	Accept		Entire input consumed and reduced to start symbol

Where do Parsing falls in the process?



Parser pseudo code: Using this Grammar $E \to (E + E) \mid NUM$

```
token = input.read()        # look-ahead token
parse_E()

func parse_E() → bool :
    switch (token)
        case NUM :
            token = input.read()
            return True
        case "(" :
            token = input.read()
            parse_E()
            if token != "+" : throw
            parse_E()
            if token != ")" : throw

            token = input.read()
            return True
        default:
            throw
```

⇒ This is Recursive-Descent Parser

That is Parser for LL(1) Grammar

LL(1) Grammar stands for: "L" : (scan) left - to - right
"L" : (produces) Left most derivation
"1" : (using) One look-ahead token

In other words, LL(1) is an instruction that say "read left to right, expand left most derivation, one token at a time"

Example: Sentence "(2 + 3)", Grammar $E \to (E + E) \mid NUM$

Read left - to - right: (→ 2 → + → 3 →)

Expand left most derivation:

1. Start with E
2. $E \to (E + E)$ // input starts with "("
3. $(E + E) \to (NUM + E)$ // expand leftmost E to NUM (lookahead "2")
4. $(NUM + E) \to (NUM + NUM)$ // expand remaining E to NUM

Non - LL(1) Grammar

One way to tell if a Grammar is non- LL(1) is that one look ahead token can match many derivations

For example:

Consider this Grammar:

$$S \rightarrow E + S \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

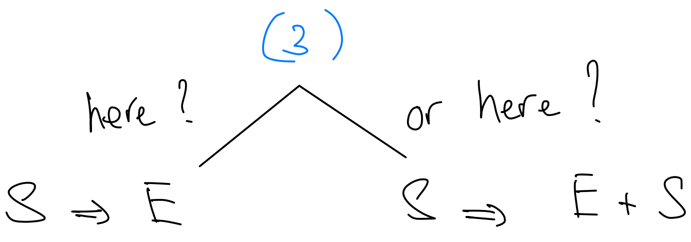
and this sentence: $(8) + 4$

We have 2 derivations:

① $S \Rightarrow E \Rightarrow (S) \Rightarrow (E) \Rightarrow (3)$

② $S \Rightarrow E + S \Rightarrow (S) + S$
$\Rightarrow (E) + S \Rightarrow (3) + E$
$\Rightarrow (3) + 4$

So, for sentence $(3) + 4$, with the same look- ahead token, I have no way to tell which production should I match



⇒ This is non- LL(1) Grammar

Convert non - LL(1) to LL(1)

Root cause in non- LL(1) Grammar is the non - decisiveness, as shown in example:

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

We can turn this into:

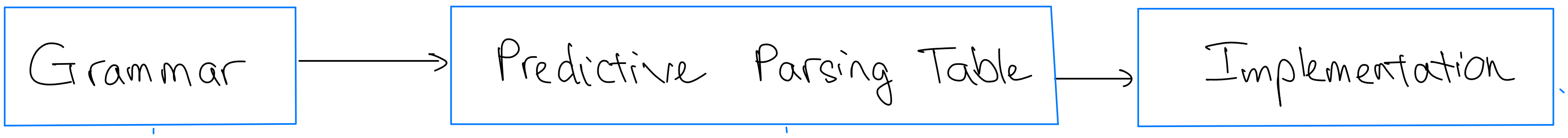
$$S \rightarrow E S'$$
$$S' \rightarrow \varepsilon$$
$$S' \rightarrow + S$$

// can think of S' as " decision maker "

So, we solve the problem of non- decisiveness in non – LL(1) with a decision maker

Recursive - Descent Parser: Implementation

General process to implement Recursive- Descent Parser:



$S \rightarrow E S'$

$S' \rightarrow \varepsilon$

$S' \rightarrow + S$

$E \rightarrow num$

$E \rightarrow (S)$

columns : Terminals

rows: non-Terminals

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

Implementations based off Parsing Table

```
void parse_S () :
    switch (token) :
        case num :
        case "(" :
            parse_E()
            parse_S' ()
            return
```

```
void parse_S' ():
    swith (token)
        case "+":
            token = input.read ()
            parse_S ()
            return
        case ")"
        case " $ " :
            return
```

```
void parse_E ():
    swith (token)
        case num:
            token = input.read ()
            return
        case "(":
            token = input.read ()
            parse_S ()
            if token != ")" : throw
            token = input.read ()
            return
```