

# MiniLang Lexer

## Complete Guide — Phase 1 of the Compiler

---

### What this guide covers:

- What a Lexer is and why it exists
- How this Lexer reads code character by character
- Every token type with examples
- The Maximal Munch rule
- How whitespace and comments are skipped
- All lexical errors the Lexer can detect
- Line and column tracking
- A full worked example you can trace by hand

**Language:** This guide is written in simple, easy English so you can read it quickly and explain it in a viva.

# 1 What is a Lexer?

A **Lexer** (also called a **Scanner** or **Tokenizer**) is the **first phase** of a compiler. Its only job is:

```
Source Code  [ LEXER ]  Tokens
```

It reads your code **character by character**, groups those characters into meaningful pieces called **lexemes**, labels each piece with a **token type**, and passes the list to the next phase (the Parser).

## 1.1 A Simple Example

Suppose your code is:

```
Let name = "abc";
```

The Lexer breaks it into:

| Lexeme | Token Type    | Value |
|--------|---------------|-------|
| Let    | LET (keyword) | —     |
| name   | IDENTIFIER    | name  |
| =      | ASSIGN        | —     |
| "abc"  | STRING        | abc   |
| ;      | SEMICOLON     | —     |

## 1.2 What the Lexer Does NOT Do

- It does **not** check if the code follows grammar rules (that is the Parser's job).
- It does **not** check types — whether you are adding a number to a string (that is the Semantic Analyzer's job).
- It does **not** run the code (that is the Interpreter's job).

**Key idea:** The code `x = + 10 5;` passes the Lexer perfectly — every character is a valid token. It is the **Parser** that rejects it, because the tokens are in the wrong order.

## 2 How the Lexer Works Step by Step

---

### Step 1 — Read Characters One by One

The Lexer keeps a **cursor** (a position marker). It starts at position 0 and moves forward one character at a time.

```
Input: Let x = 10;
      ^  cursor starts here

Read L → letter → keep reading
Read e → letter → keep reading
Read t → letter → keep reading
Read (space) → STOP → save "Let"
```

### Step 2 — Decide the Token Type

After collecting a piece of text, the Lexer checks:

- Is it a **keyword**? (Check the keyword table: Let, print, if, else)
- If not a keyword, is it a **letter/underscore**? → IDENTIFIER
- Is it a **digit**? → NUMBER
- Is it a **double quote** "?" → start reading a STRING
- Otherwise → check if it is an operator or separator

### Step 3 — Store 5 Pieces of Information

For every token the Lexer stores:

| Field  | Meaning                              | Example    |
|--------|--------------------------------------|------------|
| type   | The category of the token            | IDENTIFIER |
| lexeme | The actual text from your code       | name       |
| value  | The real value (for numbers/strings) | name       |
| line   | Which line number (starting from 1)  | 2          |
| column | Which column on that line            | 9          |

### Step 4 — Output a Flat Token List

The Lexer sends a flat list. No structure, no tree — just tokens in order, ending with a special **EOF** (End Of File) token.

```
[LET] [IDENTIFIER 'x'] [ASSIGN] [NUMBER 10] [SEMICOLON] [EOF]
```

## 3 All Token Types in MiniLang

### 3.1 Keywords (Reserved Words)

These words have special meaning. They are **case-sensitive** — must match exactly.

| Keyword | Token Type | What It Does                |
|---------|------------|-----------------------------|
| Let     | LET        | Declare a new variable      |
| print   | PRINT      | Print a value to the screen |
| if      | IF         | Start an if-condition       |
| else    | ELSE       | The else branch of an if    |

**Important:** "Let" (capital L) is a keyword. "let" (lowercase) is NOT — the Lexer would treat it as an ordinary identifier (variable name).

### 3.2 Identifiers (Variable Names)

Any name you create. Must start with a **letter or underscore**. Can contain letters, digits, and underscores after that.

Valid:     name, x, y, \_temp, total2, my\_var  
Invalid:  2total (starts with digit)  
          my-var (dash is not allowed)

**Rule in code form:**

```
[A-Za-z_][A-Za-z0-9_]*
```

### 3.3 Numbers

Whole numbers (integers) or decimals (floats).

Valid:     3, 18, 3.5, 100, 0.25  
Invalid:  3.   (no digit after dot)  
          .5   (no digit before dot)  
          12abc (letters stuck to number)

### 3.4 Strings

Text inside **double quotes**. The Lexer removes the quotes and stores the inside text as the value.

Valid:     "abc", "not adult", "hello world"  
Invalid:  "hello   (no closing quote)

**Escape Sequences** (special characters inside strings):

| Escape | Meaning         |
|--------|-----------------|
| \n     | New line        |
| \t     | Tab space       |
| \r     | Carriage return |

| Escape | Meaning                                    |
|--------|--------------------------------------------|
| \"     | A double-quote character inside the string |
| \\     | A backslash character                      |

## 3.5 Operators

Arithmetic Operators (math):

| Symbol | Token Type | Meaning                           |
|--------|------------|-----------------------------------|
| +      | PLUS       | Addition AND String Concatenation |
| -      | MINUS      | Subtraction                       |
| *      | STAR       | Multiplication                    |
| /      | SLASH      | Division                          |
| %      | PERCENT    | Modulo (remainder)                |

**Key point:** The + symbol is used for BOTH addition (3 + 5) and string concatenation ("hello " + name). The Lexer cannot tell the difference — it just outputs a PLUS token. Only the Semantic Analyzer (Phase 3) can decide, because only it knows the types of both sides.

Assignment Operator:

= → ASSIGN (put a value into a variable)

Comparison Operators:

| Symbol | Token Type | Meaning                  |
|--------|------------|--------------------------|
| ==     | EQ         | Equal to                 |
| !=     | NEQ        | Not equal to             |
| <      | LT         | Less than                |
| >      | GT         | Greater than             |
| <=     | LTE        | Less than or equal to    |
| >=     | GTE        | Greater than or equal to |

## 3.6 Separators (Punctuation)

| Symbol | Token Type | Meaning           |
|--------|------------|-------------------|
| {      | LBRACE     | Open a block      |
| }      | RBRACE     | Close a block     |
| (      | LPAREN     | Open parentheses  |
| )      | RPAREN     | Close parentheses |
| ;      | SEMICOLON  | End of statement  |

## 3.7 EOF Token

A special virtual token added at the very end. It tells the Parser: "there is nothing left to read". The Lexer always adds this.

## 4 The Maximal Munch Rule

---

This is one of the most important rules in lexer design.

**Maximal Munch:** Always read the **longest** possible token before stopping.

### Example 1: == vs = =

```
Code:    x == 5

Lexer reads: =
Looks ahead: another = ?
YES → read both as ONE token: [EQ]  ("==")

✓ CORRECT: one [EQ] token
✗ WRONG:   two [ASSIGN] tokens
```

### Example 2: <= vs < =

```
Code:    y <= 18

Lexer reads: <
Looks ahead: = after the < ?
YES → read both as ONE token: [LTE] ("<=")

✓ CORRECT: one [LTE] token
✗ WRONG:   [LT] then [ASSIGN]
```

### Example 3: != vs ! (error)

```
Code:    a != b

Lexer reads: !
Looks ahead: = after the ! ?
YES → read both as ONE token: [NEQ]  ("!=")

But if just: !
No = after it → ERROR
"Unexpected character '!' – did you mean '!='?"
```

Without maximal munch, the Lexer would break multi-character operators into pieces, and the Parser would get confused.

## 5 Whitespace and Comments

---

### 5.1 Whitespace

Spaces, tabs, and newlines are **skipped**. They only serve to separate tokens.

```
Input:  Let    x    =    10 ;  
Output: [LET] [IDENTIFIER] [ASSIGN] [NUMBER] [SEMICOLON]  
        (all spaces gone)
```

### 5.2 Line Comments //

Everything after `//` until the end of the line is ignored.

```
Input:  Let x = 10;  // this is a comment  
Output: [LET] [IDENTIFIER] [ASSIGN] [NUMBER] [SEMICOLON]  
        (comment ignored)
```

### 5.3 Block Comments /\* ... \*/

Everything between `/*` and `*/` is ignored. Can span multiple lines.

```
Input:  Let x = /* comment */ 10;  
Output: [LET] [IDENTIFIER] [ASSIGN] [NUMBER] [SEMICOLON]  
        (block comment ignored)
```

## 6 Lexical Error Detection

---

The Lexer detects errors — pieces of code that cannot form any valid token. When it finds one, it **records the error** (with line and column) and **keeps scanning**, so one mistake does not hide the rest.

### 6.1 Unterminated String

```
Code:  Let s = "hello;
```

**Error:** Unterminated string literal (missing closing "")

**Why:** The Lexer expects a closing " but reaches end of line.

### 6.2 Invalid Number Literal

```
Code:  Let a = 12abc;
```

**Error:** Invalid number literal '12abc'

**Why:** A number cannot have letters attached. 12 is a number, abc is an identifier, 12abc is neither.

### 6.3 Malformed Number

```
Code:  Let x = 3.;
```

**Error:** Malformed number literal: digit expected after '.'

**Why:** A decimal point must be followed by at least one digit.

### 6.4 Unexpected Character

```
Code:  Let x = 5 @ 3;
```

**Error:** Unexpected character '@'

**Why:** The @ symbol is not part of MiniLang's alphabet.

### 6.5 Stray Exclamation Mark

```
Code:  if (a ! b) print x;
```

**Error:** Unexpected character '!' (did you mean '!=')?

**Why:** The ! must be followed by = to form !=.

### 6.6 Unterminated Block Comment

```
Code:  /* comment starts  
       but never closes
```

**Error:** Unterminated block comment (missing '/')

**Why:** Every /\* must have a matching \*/.



## 7 Line and Column Tracking

---

The Lexer records the **line number** and **column number** where every token starts. This is important because when there is an error, you need to know **exactly where** in the code it happened.

```
Line 1: {
Line 2:     Let name = "abc";
           ^   ^       ^
           col5 col9  col16

Token table:
LET      'Let'    @line 2, col 5
IDENTIFIER 'name'  @line 2, col 9
ASSIGN   '='      @line 2, col 14
STRING   '"abc"'  @line 2, col 16
```

When an error occurs, the message says exactly where to look:

```
Lexical error [line 2, col 16]: Unterminated string

This tells you:
→ look at line 2
→ look at column 16
→ the problem is right there
```

## 8 Full Worked Example (Trace by Hand)

Here is a complete MiniLang program and its token output:

```
{
  Let name = "abc";
  Let y = 3;
  print y;
  y = 3 * 2 + 5;
  print name;
  if (y < 18) print "not adult";
}
```

Complete Token Table:

| #  | Type       | Lexeme | Value | Line:Col |
|----|------------|--------|-------|----------|
| 1  | LBRACE     | {      | —     | 1:1      |
| 2  | LET        | Let    | —     | 2:5      |
| 3  | IDENTIFIER | name   | name  | 2:9      |
| 4  | ASSIGN     | =      | —     | 2:14     |
| 5  | STRING     | "abc"  | abc   | 2:16     |
| 6  | SEMICOLON  | ;      | —     | 2:21     |
| 7  | LET        | Let    | —     | 3:5      |
| 8  | IDENTIFIER | y      | y     | 3:9      |
| 9  | ASSIGN     | =      | —     | 3:11     |
| 10 | NUMBER     | 3      | 3     | 3:13     |
| 11 | SEMICOLON  | ;      | —     | 3:14     |
| 12 | PRINT      | print  | —     | 4:5      |
| 13 | IDENTIFIER | y      | y     | 4:11     |
| 14 | SEMICOLON  | ;      | —     | 4:12     |
| 15 | IDENTIFIER | y      | y     | 5:5      |
| 16 | ASSIGN     | =      | —     | 5:7      |
| 17 | NUMBER     | 3      | 3     | 5:9      |
| 18 | STAR       | *      | —     | 5:11     |
| 19 | NUMBER     | 2      | 2     | 5:13     |
| 20 | PLUS       | +      | —     | 5:15     |
| 21 | NUMBER     | 5      | 5     | 5:17     |
| 22 | SEMICOLON  | ;      | —     | 5:18     |
| 23 | PRINT      | print  | —     | 6:5      |

| #  | Type       | Lexeme      | Value     | Line:Col |
|----|------------|-------------|-----------|----------|
| 24 | IDENTIFIER | name        | name      | 6:11     |
| 25 | SEMICOLON  | ;           | —         | 6:15     |
| 26 | IF         | if          | —         | 7:5      |
| 27 | LPAREN     | (           | —         | 7:8      |
| 28 | IDENTIFIER | y           | y         | 7:9      |
| 29 | LT         | <           | —         | 7:11     |
| 30 | NUMBER     | 18          | 18        | 7:13     |
| 31 | RPAREN     | )           | —         | 7:15     |
| 32 | PRINT      | print       | —         | 7:17     |
| 33 | STRING     | "not adult" | not adult | 7:23     |
| 34 | SEMICOLON  | ;           | —         | 7:34     |
| 35 | RBRACE     | }           | —         | 8:1      |
| 36 | EOF        | <eof>       | —         | 9:1      |

**Total tokens: 36 | Lexical errors: 0**

## 9 Tracing a Single Line Step by Step

---

Let us trace line 5: `y = 3 * 2 + 5;`

| Step | Character | Action                                  | Token Produced   |
|------|-----------|-----------------------------------------|------------------|
| 1    | y         | Letter → read identifier → 'y'          | IDENTIFIER 'y'   |
| 2    | (space)   | Skip whitespace                         | —                |
| 3    | =         | Look ahead: next char is space, not '=' | ASSIGN '='       |
| 4    | (space)   | Skip whitespace                         | —                |
| 5    | 3         | Digit → read number → '3'               | NUMBER (value 3) |
| 6    | (space)   | Skip whitespace                         | —                |
| 7    | *         | Single-char operator                    | STAR '*'         |
| 8    | (space)   | Skip whitespace                         | —                |
| 9    | 2         | Digit → read number → '2'               | NUMBER (value 2) |
| 10   | (space)   | Skip whitespace                         | —                |
| 11   | +         | Single-char operator                    | PLUS '+'         |
| 12   | (space)   | Skip whitespace                         | —                |
| 13   | 5         | Digit → read number → '5'               | NUMBER (value 5) |
| 14   | ;         | Separator                               | SEMICOLON ';'    |

Notice how the Lexer does not know that `*` should happen before `+`. It just outputs a flat list. Deciding that `3 * 2` groups together and `+ 5` is separate — that is the **Parser's** job.

## 10 How Keywords Are Recognized

The Lexer uses a simple two-step trick:

**Step 1:** When it sees a letter, it reads the whole word as if it were an identifier.

**Step 2:** After reading the whole word, it looks it up in a **keyword table**:

```
Keyword Table:
"Let"    → LET
"print"  → PRINT
"if"     → IF
"else"   → ELSE
```

If the word is in the table → it is a **keyword token**.

If it is NOT in the table → it is an **identifier token**.

**Example:**

```
"Let"    → found in table → LET token
"name"   → not in table   → IDENTIFIER token
"let"    → not in table   → IDENTIFIER token (case matters!)
"Print"  → not in table   → IDENTIFIER token (capital P)
```

This is called the **"identifier first, then keyword lookup"** technique. It is the standard method used in real compilers and keeps the scanner simple.

## 11 Summary

| Question                      | Answer                                                  |
|-------------------------------|---------------------------------------------------------|
| What does the Lexer do?       | Reads code character by character and produces tokens   |
| What is a lexeme?             | The actual piece of text from the source code           |
| What is a token?              | A lexeme + its category label (type)                    |
| What is a token type?         | The category: KEYWORD, IDENTIFIER, NUMBER, STRING, etc. |
| What is Maximal Munch?        | Always read the longest possible token first            |
| Does the Lexer check grammar? | NO — that is the Parser's job                           |
| Does the Lexer check types?   | NO — that is the Semantic Analyzer's job                |
| What does EOF mean?           | End Of File — a virtual token marking the end of input  |
| How are errors handled?       | Recorded with line/column, scanning continues           |

**The complete compiler pipeline:**

```
Source Code
  |
[1] Lexer      → Tokens      ← YOU ARE HERE
  |
[2] Parser     → AST
```

```
|  
[3] Semantic Analysis → Checked AST  
|  
[4] Interpreter      → Program Output
```

---

**End of MiniLang Lexer Guide**