COMPILER CONSTRUCTION

**Input and Output of a Mini Compiler**

The **input** to a mini compiler is source code written in a high-level programming language or a simplified subset of one. The **output** varies depending on the stage of compilation and the purpose of the compiler. It can be tokens, an Abstract Syntax Tree (AST), intermediate representation (IR), optimized IR, or machine code.

### ****1. Input****

The input to the mini compiler is typically a source code string.

**Example Input (Source Code)**:

int x = 10;

int y = x + 20;

if (y > 15)

{

y = y \* 2;

}

return y;

### ****2. Output*1:*****

### ****Lexical Analysis (Tokens)****

The first stage outputs tokens that represent the smallest meaningful elements of the source code.

**Example Output (Tokens)**:

[

{ Type: "Keyword", Value: "int" },

{ Type: "Identifier", Value: "x" },

{ Type: "Operator", Value: "=" },

{ Type: "Number", Value: "10" },

{ Type: "Symbol", Value: ";" },

{ Type: "Keyword", Value: "int" },

{ Type: "Identifier", Value: "y" },

{ Type: "Operator", Value: "=" },

{ Type: "Identifier", Value: "x" },

{ Type: "Operator", Value: "+" },

{ Type: "Number", Value: "20" },

{ Type: "Symbol", Value: ";" },

{ Type: "Keyword", Value: "if" },

{ Type: "Symbol", Value: "(" },

{ Type: "Identifier", Value: "y" },

{ Type: "Operator", Value: ">" },

{ Type: "Number", Value: "15" },

{ Type: "Symbol", Value: ")" },

{ Type: "Symbol", Value: "{" },

{ Type: "Identifier", Value: "y" },

{ Type: "Operator", Value: "=" },

{ Type: "Identifier", Value: "y" },

{ Type: "Operator", Value: "\*" },

{ Type: "Number", Value: "2" },

{ Type: "Symbol", Value: ";" },

{ Type: "Symbol", Value: "}" },

{ Type: "Keyword", Value: "return" },

{ Type: "Identifier", Value: "y" },

{ Type: "Symbol", Value: ";" }

]

#### **Stage 2: Syntax Analysis (Abstract Syntax Tree)**

The second stage constructs a hierarchical representation of the program, often as an Abstract Syntax Tree (AST).

**Example Output (AST)**:

Program

├── Declaration (int x = 10)

├── Declaration (int y = x + 20)

├── IfStatement

│ ├── Condition (y > 15)

│ ├── Block

│ │ └── Assignment (y = y \* 2)

├── ReturnStatement (y)

#### **Stage 3: Optimization**

The optimizer simplifies the code by applying techniques such as constant folding or dead code elimination.

**Example Output (Optimized AST)**:

Program

├── Declaration (int x = 10)

├── Declaration (int y = 30) // x + 20 evaluated at compile time

├── IfStatement

│ ├── Condition (true) // y > 15 evaluated at compile time

│ ├── Block

│ │ └── Assignment (y = 60) // y \* 2 evaluated at compile time

├── ReturnStatement (60)

#### **Stage 4: Code Generation**

The final stage generates low-level code (e.g., assembly, machine code) or an intermediate representation (IR).

**Example Output (Pseudo-Assembly Code)**:

LOAD 10 INTO R1 // x = 10

ADD R1, 20 INTO R2 // y = x + 20

IF R2 > 15 JUMP TO L1

MULTIPLY R2, 2 INTO R2 // y = y \* 2

L1: RETURN R2

## Summary Table

| **Stage** | **Input** | **Output** |
| --- | --- | --- |

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| Stage | Input | Output |
| Lexical Analysis | Source Code | List of Tokens |
| Syntax Analysis | Tokens | Abstract Syntax Tree (AST) |  |
| Optimization | AST | Optimized AST |
| Code Generation | |  | | --- | | Optimized AST |  |  | | --- | |  | | Machine Code or Intermediate Representation |