Republic of the Philippines
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
College of Computer and Information Sciences
Sta.Mesa, Manila

Programming Language Proposal

Prof. Ria Ambrocio Sagum

In Partial Fulfillment of the Requirement for:
COSC 303 | Principles of Programming Languages

By:
BSCS 3-4 | Group 6

Members:
Agasino, Gale Franchesca
Apolonia, Madelyn D.
Bhasa, Rein Cherztin
Bondad, Felicity Joy S.
Gaa, Angelo Gabrielle
Gonzales, Eujen C.
Leñar, Jorelle Cybee
Silvor, Cyrus Joseph C.

October 2025

Contents

I. Introduction

PyC is a procedural programming language that combines the structural strictness of C with the readability and simplicity of Python. It aims to make a place that is easy to get to and works well for both new and experienced programmers. The language is designed to make programming concepts easier to grasp while preserving the logical and structural foundations of modern languages, making it suitable for both educational use and practical application.

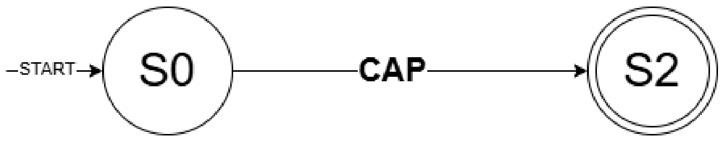
The idea of PyC comes from the ideologies of C and Python, which are similar yet also very different. C is known for its efficiency, performance, and low-level control, while Python is praised for its simplicity, readability, and speed of development. PyC wants to find a balance between clarity, control, and ease of learning by combining these features. Its primary goal is to provide a language that simplifies the learning process without sacrificing logical rigor or expressiveness, serving as both an entry-level tool and a foundation for understanding more advanced programming paradigms.

II. Syntactic Elements of Language

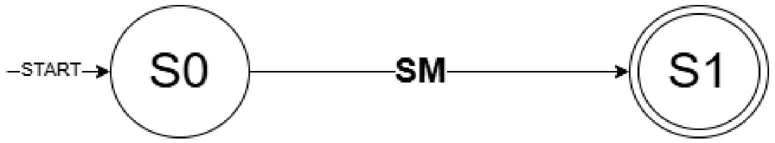
1. Character Set

These are all the valid characters in our language.
ALPHA = {CAP, SM, DIGITS, SYMBOLS}

- CAP = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
  Machine:

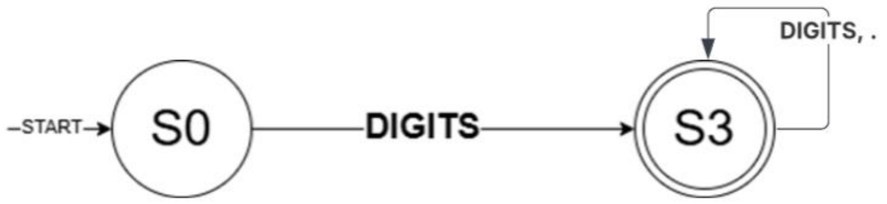


- SM = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}
  Machine:

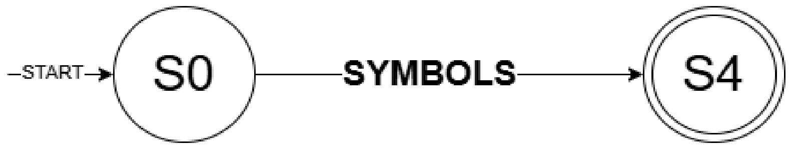


- DIGITS = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

  Machine:



- SYMBOLS = {` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ - | / \ ; : ' " , . ?}



2. Identifiers

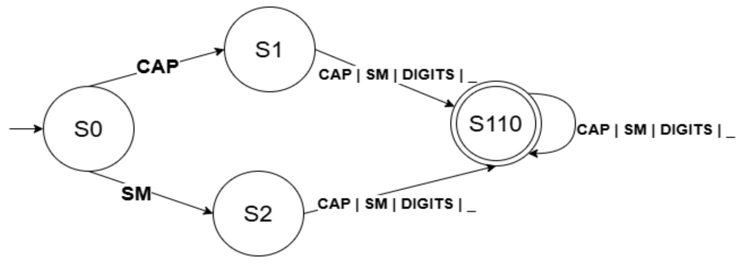
Identifiers are names given to specific elements of a program. They help make the code easy to understand and organize by giving meaningful names to data and operations. These are the rules for naming identifiers in this programming language:

1. An identifier must begin with a letter (CAP or SM).
2. The succeeding characters may consist of letters, digits, or underscores.
3. An identifier cannot begin with a digit.
4. Reserved words or keywords cannot be used as identifiers.
5. Special symbols such as @, #, $, or spaces are not allowed.
6. Identifiers are case-sensitive, meaning Value and value are considered different.

Regular Expression:

Identifier = (CAP ∪ SM) · (CAP ∪ SM ∪ DIGITS ∪ _)*

Machine:



3. Operation Symbols

The language builds upon C's minimalist operations and efficiency by adding Python's high-level of functionality and clarity.

a) Arithmetic Operations

1.) **Addition (+)**
   - Adds two numerical operands (INT, FLOAT).
   - No string/MAP concatenation.

2.) **Subtraction (-)**
   - Subtracts one numerical operand from another.

3.) **Multiplication (*)**
   - Multiplies two numerical operands. No sequence repetition.

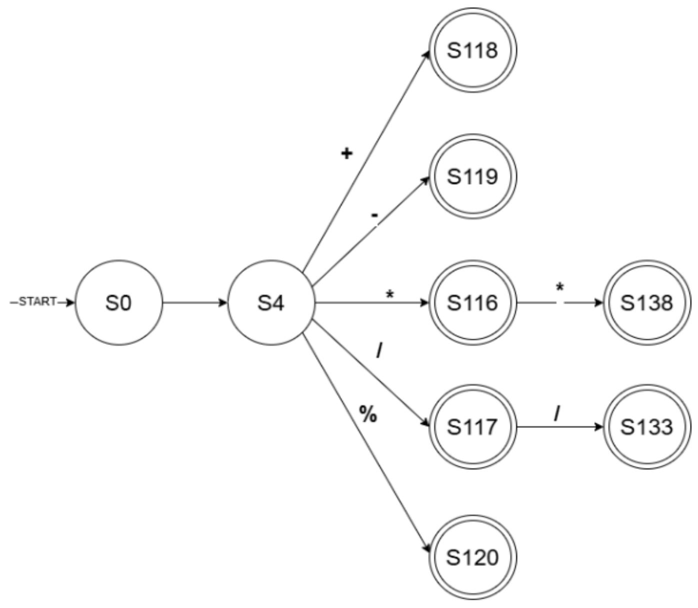
4.) **Division (/)**
   - Performs true division, always resulting in a FLOAT type.

5.) **Modulo (%)**
   - Returns the remainder of the division.
   - Must be used only with INT operands.

6.) **Exponentiation (**)**
   - Raises the first operand to the power of the second operand.

7.) **Integer Division (//)**
   - Keeps the whole number part of the division, it discards (or "truncates") the decimal or fractional part.

Machine:

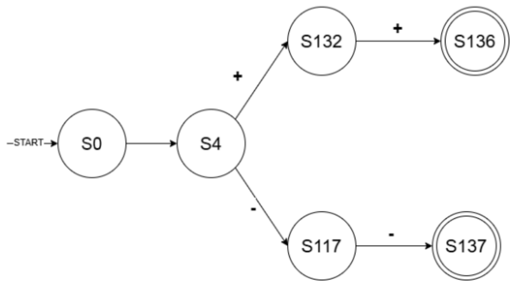


b) Unary Operations

1.) **Increment (++)**
- Add by 1
- Postfix and Prefix

2.) **Decrement (--)**
- Subtract by 1
- Postfix and Prefix

Machine:



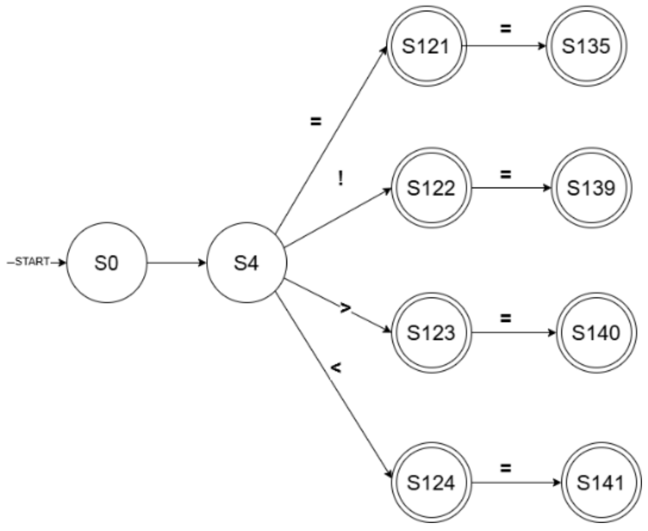
c) Assignment Operator (=)

d) Boolean Operations

d.1) Relational

1. **Equal To (==)**
2. **Not Equal To (!=)**
3. **Greater Than (>)**
4. **Less Than (<)**
5. **Greater Than Or Equal To (>=)**
6. **Less Than Or Equal To (<=)**

Machine:

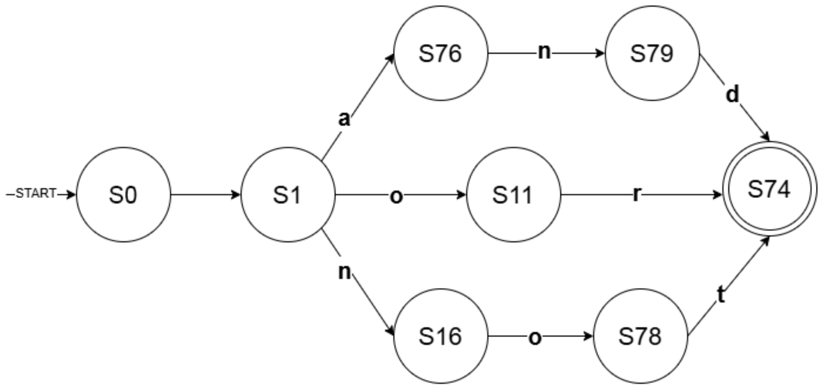


Relational and Assignment Operator

d.2) Logical

1. **Logical AND (and)**
2. **Logical OR (or)**
3. **Logical NOT (not)**

Machine:



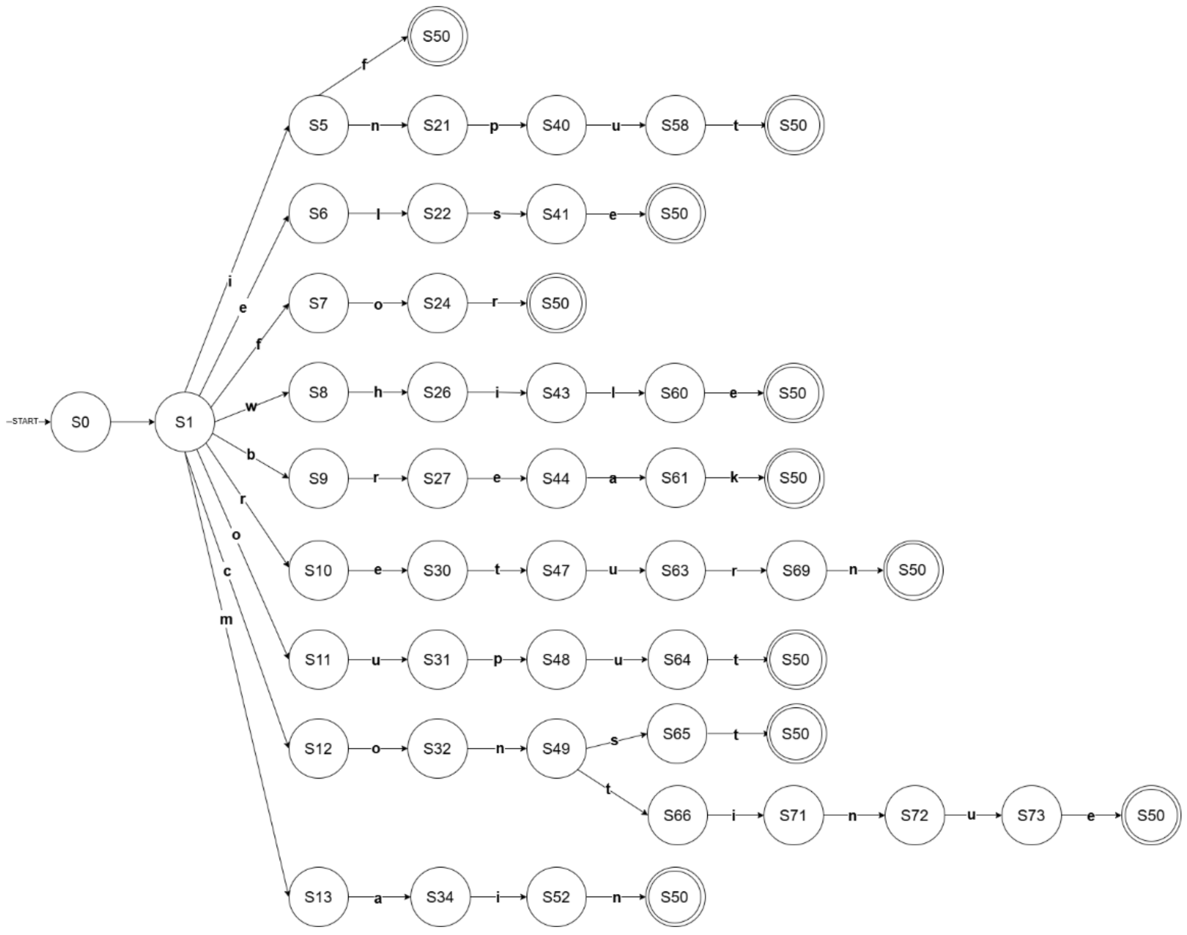
4. Keywords and Reserved Words

a. Keywords

Keywords and reserved words are words with special meanings to the compiler. All keywords are reserved words, but not all reserved words are keywords.
The following are the keywords in our language:

1. **if –** It is the first keyword in decision-making.
2. **else –** It is used when the condition within if is not met.
3. **for** – It is used for repeating a block of code until a condition is met.
4. **while** - It is used for repeating a block of code as long as a condition is true.
5. **break** – It is used for terminating a loop immediately.
6. **continue** – It is used within loops to skip the current iteration and proceed with the next iteration.
7. **return** – It is used to terminate a function and return a value.
8. **input** – It is used for user input.
9. **output** – It is used to display text.
10. **const** – It is used to declare an unchangeable variable.
11. **main** - mark as the entry point for program execution

Machine:

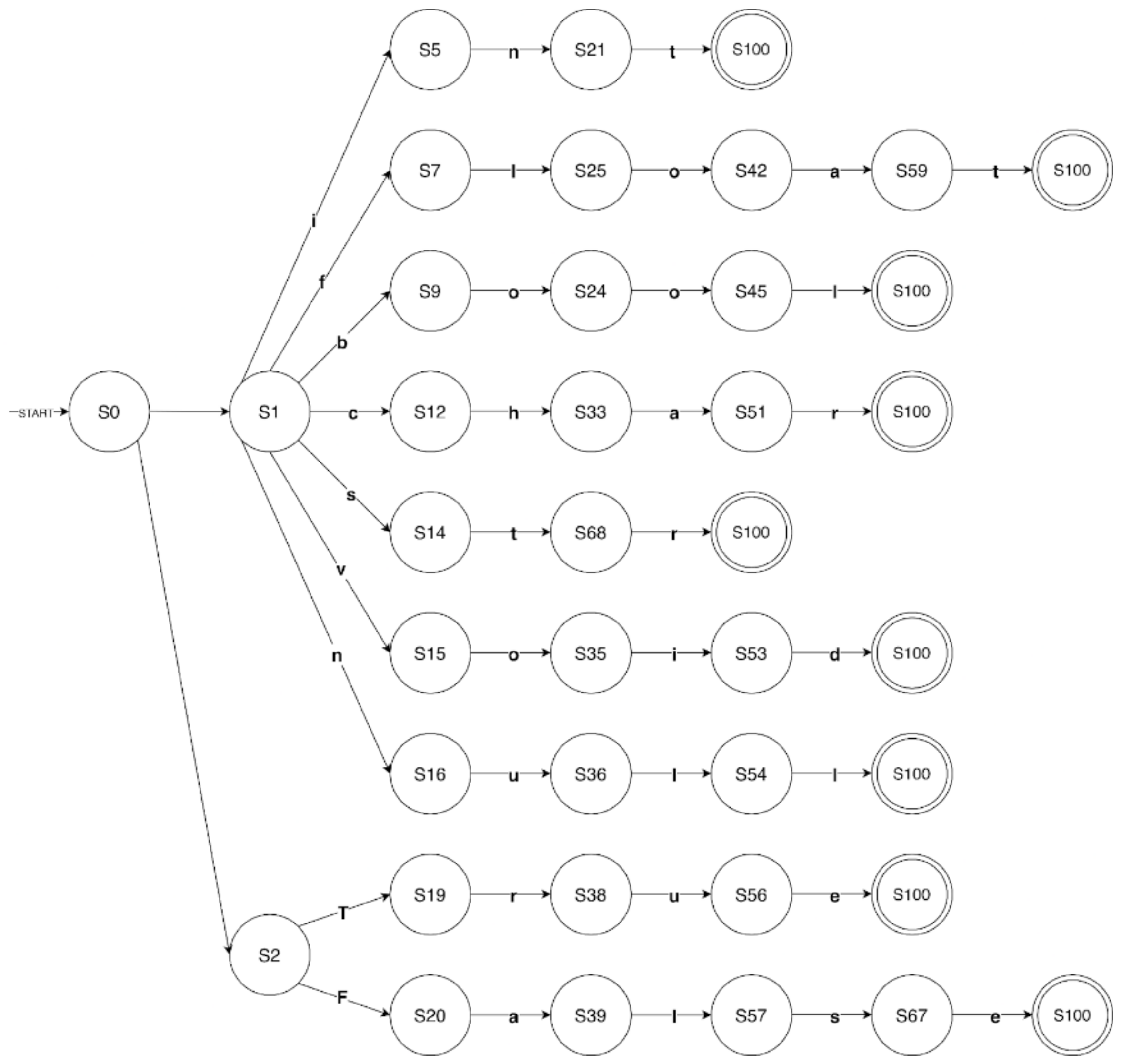

START
S0
S1
i
e
f
w
b
r
o
c
m
S5 f S50
S5 n S21 p S40 u S58 t S50
S6 l S22 s S41 e S50
S7 o S24 r S50
S8 h S26 i S43 l S60 e S50
S9 r S27 e S44 a S61 k S50
S10 e S30 t S47 u S63 r S69 n S50
S11 u S31 p S48 u S64 t S50
S12 o S32 n S49 s S65 t S50
S49 t S66 i S71 n S72 u S73 e S50
S13 a S34 i S52 n S50

b. Reserved Words

Reserved words are identifiers that are set aside for use by the language and cannot be used as variable names, function names, or any other identifier. This set primarily includes the fundamental data types and Boolean literals.

1. **int** - Integer data type representing whole numbers.
2. **float** - Floating-point number data type for numbers with decimal points.
3. **bool** - Boolean data type that holds True or False.
4. **char** - Character data type.
5. **str** - String data type.
6. **void** - Used to declare a function that does not return a value.
7. **True** - Boolean literal representing the true value.
8. **False** - Boolean literal representing the false value.
9. **null** - Check if the variable is empty or holds no value.

Machine:



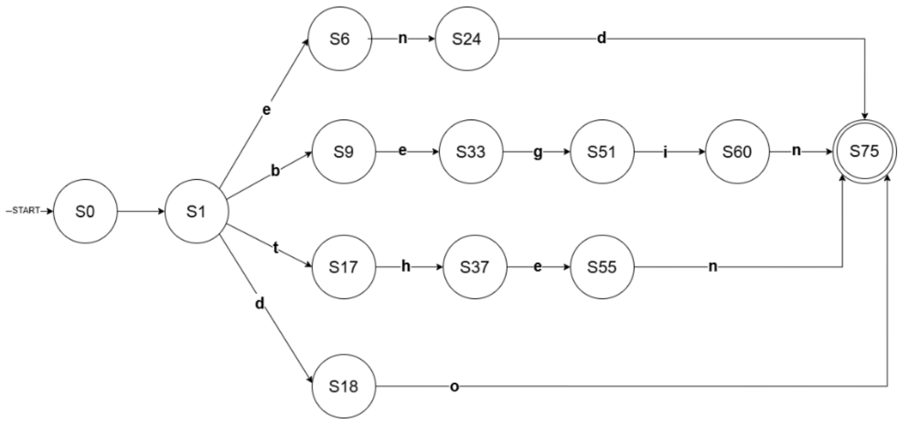
5. Noise Words

PyC supports a small set of optional "noise words" to improve readability and natural flow of code. These words do not affect execution but make the language more descriptive for beginners. Examples include then, do, end, and begin.

- **then**
  Used after "if" conditions. It doesn’t affect the logic of the syntax but it makes it more readable.
  Example:
  if (<condition>) **then** { <statements> }

- **begin**
  Can be put at the start of a block
  Example:
  main() begin {
  output("Start!"); }
- **end**
  could optionally mark block endings
  Example: if (x == 5) { output("ok"); } end
- **do**
  used before loop body
  Example: while (<condition>) do { <statements> }

Machine:



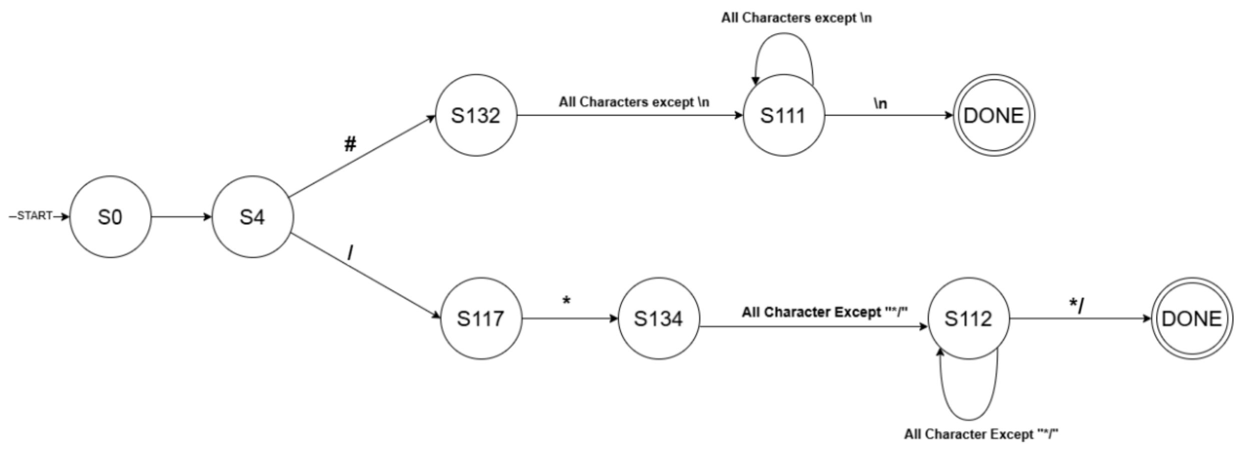
6. Comments

Comments are used to provide notes, explanations, or reminders within the code. They are ignored by the compiler or interpreter and serve solely for the programmer's reference.
Two styles of comments are supported for flexibility and familiarity:

- **Single-line comments:** Begin with #
  Syntax:
  # This is a single-line comment
- **Multi-line comments:** Enclosed between /* and */
  Syntax:
  /*
  This is a multi-line comment. It can span multiple lines.
  */

This dual style commenting approach allows programmers to use whichever format they are more comfortable with, enhancing accessibility and convenience.

Machine:



7. Blanks/Spaces

Blanks and spaces in the language are primarily used for readability and do not affect the logic or output of the program. They serve to separate tokens and improve the overall clarity of the code.

Rules for Blanks:

1. Multiple spaces are treated as a single space.
2. Spaces must separate identifiers, keywords, and operators.
3. Indentation is optional but recommended for better readability.
4. Spaces within string literals are preserved.

Syntax:
```
<identifier> = <value1> <operator> <value2>;
```

Sample:
```
total = price * quantity;
if (total > 1000) {
  output("Discount applied");
}
```

Blanks allow flexibility in formatting, helping maintain an organized and readable program structure.

8. Delimiters & Brackets

The proposed language adopts C-style delimiters and brackets to make the structure of the program clear and organized. These symbols group related lines of code and define boundaries for functions, loops, and conditions.

Delimiters Used:

- ( ) – for expressions or conditions
- { } – for code blocks
- [ ] – for array indexing or lists
- ; – marks the end of a statement
- , – separates elements or parameters

Syntax:

```
( ) → if (<condition>) { <statement>; }
{ } → { <code block> }
[ ] → array[<index>]
;  → end of <statement>;
,  → output(<item1>, <item2>);
```

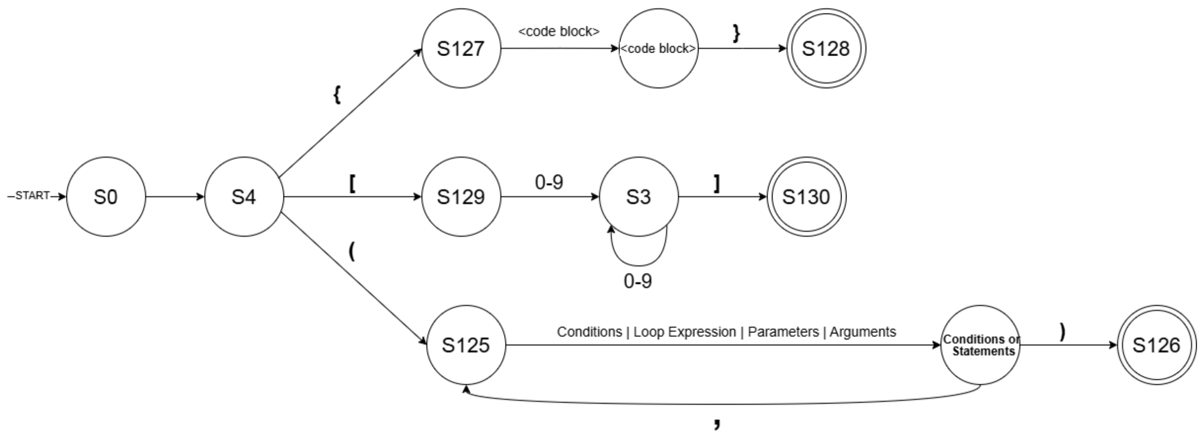
Sample:

```
numbers[3] = {10, 20, 30};

void showNumbers(int num) {
  for (i = 0; i < 3; i++) {
    output("Number {i+1} is {numbers[i]}"); }
}

if (numbers[0] > 5) {
  showNumbers(numbers[0]);
}
```

Machine:



This structure makes it easy for programmers to understand how the code flows while keeping it visually organized. The C-style braces { } help group related lines together, improving readability and reducing confusion in longer programs.

9. Free-and-Fixed-Field Formats

The language follows a free-field format, allowing flexibility in how programmers arrange their code. There are no restrictions on where statements begin in a line.
Characteristics:

- Code may start in any column.
- Indentation is optional but encouraged.
- Statements can span multiple lines if enclosed properly.

Syntax:
```
if (<condition>) {
  <statement>;
} else {
  <statement>;
}
```

Sample:
```
if (score >= 75) {
  output("Passed");
}
else {
  output("Failed");
}
```

This format promotes readability while maintaining structure, allowing programmers to freely style their code.

10. Expression

Expressions are values, variables, operators, and functions put together and computed by the computer to come up with a result. They are the computational elements of a program where real calculations, comparisons, or logical evaluations take place.

a) Mathematical/Arithmetic Expression

In our programming language, mathematical/arithmetic expressions are used to carry out calculations with numeric values and variables. The expressions can consist of arithmetic operators like addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). Parentheses () can be applied to enclose parts of an expression and dictate the evaluation order. The outcome of an arithmetic expression is a single numeric value that can either be placed in a variable or be used as part of other expressions.

Syntax:
```
<data_type> <variable> = <arithmetic_expression>;
<arithmetic_expression> → <operand> <operator> <operand>
<operator> → + | - | * | / | %
```

Sample:
int result = (a + b) * 2 - c / d;

In this case, the arithmetic expression (a + b) * 2 - c / d is calculated based on operator precedence, and the result is assigned to the variable result.

Precedence Level	**Operator(s)**	**Description**	**Associativity**
1	()	Parentheses (used to group expressions and alter the natural order of evaluation)	Left to Right
2	*, /, %	Multiplication, Division, Modulus	Left to Right
3	+, -	Addition, Subtraction	Left to Right

Rules of Evaluation:
1. Parentheses-enclosed expressions are evaluated first.
2. Among the remaining operations, multiplication, division, and modulus are evaluated before addition and subtraction.
3. When operators have the same precedence, the operation is carried out from left to right.

b) Boolean Expression (Relational and Logical)

Boolean expressions are used to evaluate value by comparing operands or checking logical relationships to determine whether a statement is true or false. The expressions can consist of relational and logical operators like equal to (==), not equal (!=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), logical Not (not), logical And (and), and logical Or (or). The expressions could be grouped or evaluated in order within the parenthesis operator () making it into a condition. The result for a Boolean expression is a Boolean value—either True or False— which can be used to make decisions or control the flow of the program.

Syntax: (for all boolean operators except NOT)
```
<data_type> <variable> = <value>;
if (<condition>){
  <statement>;
}
<condition> → <operand> <operators>
<operand>
```

Syntax: (for NOT operator)
```
<data_type> <variable> = <value>;
if (<condition>){
  <statement>;
}
<condition> → <operator> <operand>
<operator> → not
```

<operator> → ==, !=, >, <, >=, <=, and, or
<operand> → <variable> | <value>

<operand> → <variable> | <expression>

Sample: (for all boolean operators except NOT)

```
int age = 10;
if (age >= 18)
   output("You are an adult.");
}
```

Sample: (for NOT operator)

```
int online = 0;
if (not online) {
   output("You are offline.");
}
```

Like arithmetic expression, boolean expressions are evaluated based on its operator precedence and the result is assigned to the variable result.

Precedence Level	**Operator(s)**	**Description**	**Associativity**
1	()	Parenthesis for grouping expression, especially for constructing conditions	Left to Right
2	==, !=, >, <, >=, <=	(Relational) Equal To, Not Equal, Greater Than, Less Than, Greater Than or Equal To, Less Than or Equal To	Left to Right
3	not	(Logical) Not	Right To Left
4	and	(Logical) And	Left to Right
5	or	(Logical) Or	Left to Right

Rules of Evaluation

1. Expressions within the parenthesis should be evaluated first.
2. Higher precedence are evaluated before the lower precedence.
3. All of the boolean operators will be left-associative except for logical Not operator.

11. Declaration Statement

Declaration statements in our programming language are used to define new variables, constants, or data structures before they are used in the program. A declaration specifies the variable name, its data type, and optionally its initial value.

- **Simple declaration**
  It introduces a variable with a specific data type without assigning any initial value.

  Syntax:
  <data_type> <variables>;

  Sample:
  int num;
  bool isTrue;

- **Initialized declaration**
  It defines a variable and assigns it an initial value in a single statement.

  Syntax:
  <data_type> <variables> = <value>;

  Sample:
  int num = 10;

a) Assignment Statements

Assignment statements in our programming language are used to **store or update values** in variables. The assignment operator = assigns the value on the right-hand side to the variable on the left-hand side.

There are three main forms of assignment in our language:

• Simple assignment:
A simple assignment statement assigns a single value to a single variable. The variable must be declared with a valid data type before or during assignment.

• Array assignment:
Array assignments are used to initialize a collection of elements under a single variable name. Elements can be of any valid data type supported by the language.

Syntax:
<data_type> <variable> = <value>;

Sample:
int x = 10;
bool y = True;

Syntax:
<data_type> <array_name>[] = {<value1>, <value2>, ...};
<data_type> <array_name>[<size>] = {<value1>, <value2>, ...};

Sample:
char array[] = {"a", "b", "c"};
str array[2] = {"Hello", "World!"};

• **Multiple assignment:**
Multiple variables of the same data type can be declared and initialized in a single line. This improves readability and reduces redundancy in variable declarations.

Syntax:
<data_type> <variable1> = <value1>, <variable2> = <value2>, ... ;

Sample:
int a = 5, b = 7;

Rules:

1. Each assignment statement is required to end with a semicolon (;).
2. Data types must match the assigned values.
3. Strings must be enclosed in double quotes (").
4. Boolean values are case-sensitive (True, False).
5. Arrays automatically infer their size if not explicitly declared.

b) Conditional Statements

Conditional statements control the flow of execution based on conditions. They evaluate Boolean expressions (True or False) and decide which block of code to run.

1. Simple If Statement
Executes a block of code only if the condition is True.

Syntax:
```
if (<condition>){
   <statements>
```

2. If–Else Statement
Provides an alternative block of code that executes when the condition is False.

Syntax:
```
if (<condition>){
   <statements_if_true>
}
```

Sample:

```
if (age <= 3){
  output("Toddler");
}
```

```
}
else{
  <statements_if_false>
}
```

Sample:

```
if (age <= 3){
  output("Toddler");
}
else{
  output("Older than toddler age");
}
```

3. Else–If Ladder

Used to check multiple conditions in sequence. The first True condition executes its corresponding block, and the rest are skipped.

Syntax:

```
if (<condition1>){
  <statements>
}
else if (<condition2>){
  <statements>
}
else{
  <statements>
}
```

Sample:

```
if (age <= 3){
  output("Toddler");
}
else if ((age > 3) and (age <= 12)){
  output("Kid");
}
else if ((age > 12) and (age < 20)){
  output("Teenager");
}
else{
```

4. Nested Conditional Statements

A nested conditional statement is a conditional structure placed inside another conditional block.
It allows more precise decision-making when multiple levels of conditions need to be checked.

Syntax:

```
if (<condition1>){
  if (<condition2>){
    <statements_if_both_true>
  }
  else{
    <statements_if_first_true_second_false>
  }
}
else{
  <statements_if_first_false>
}
```

Sample:

```
if (age >= 0){
  if (age <= 3){
    output("Toddler");
  }
  else if ((age > 3) and (age <= 12)){
    output("Kid");
  }
  output("Adult");
}
```

```
  else if ((age > 12) and (age < 20)){
    output("Teenager");
  }
  else{
    output("Adult");
  }
}
else{
  output("Invalid age");
}
```

Rules

1. Conditions must evaluate to True or False.
2. Logical operators include "and" and "or".
3. Parentheses around conditions are required for clarity.
4. Each block of statements must be enclosed in curly braces {}.
5. else if statements are evaluated in order, once a condition is True, the rest are skipped.
6. Each nested if must be enclosed within the braces {} of its parent condition.
7. Indentation (optional) can be used for readability but is not required for correctness.

c) Iterative Statements

Loops allow repetition of code until a condition changes or a sequence ends. Our programming language supports two main types of loops: **while loops** and **for loops**, both of which can be nested.

1. While Loop

The while loop repeatedly executes a block of code as long as the given condition remains True. The condition is evaluated before each iteration.

Syntax:

```
while (<condition>){
  <statements>
}
```

Sample:

```
int count = 0;
while (count < 3){
  output("{count}");
  count = count + 1;
}
```

2. Nested While Loop

A nested while loop is a loop placed inside another while loop. The inner loop executes completely for each iteration of the outer loop.

Syntax:

```
while (<outer_condition>){
   while (<inner_condition>){
     <statements>
   }
}
```

Sample:

```
int x = 1;
while (x <= 3){
 int y = 1;
```

```
    while (y <= 2){
      output("Outer: {x}, Inner: {y}");
      y = y + 1;
    }
    x = x + 1;
}
```

3. For Loop
The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and update.

Syntax:
```
for (<initialization>; <condition>; <update>){
   <statements>
}
```

Sample:
```
for (int i = 0; i < 5; i++){
   output("{i}");
}
```

4. Nested For Loop
A nested for loop places one for loop inside another. The inner loop completes all its iterations for every iteration of the outer loop.

Syntax:
```
for (<initialization>; <condition>; <update>) {
  for (<init2>; <condition2>; <update2>) {
    <statements>
  }
}
```

Sample:
```
for (int i = 1; i <= 3; i++){
   for (int j = 1; j <= 2; j++){
     output("i: {i}, j: {j}");
   }
}
```

d) Input/Output Statement

Input and output statements in our programming language are used to communicate between the user and the program. These statements handle **reading user input and displaying program output** in a clear and readable format.

1. Output Statement

Syntax
```
output("<string or value>");
```

Sample:
```
output("Hello, World!");
output("The result is {result}");
output("{a + b}");
```

2. Input Statement

Syntax

```
<variable> = input(<prompt>);
```

Sample:

```
str name = input("Enter your name: ");
int age = int(input("Enter your age: "));
output("Hello {name}, you are {age} years old!");
```

Rules:

1. Expressions inside {} can include variables or arithmetic operations.
2. Strings without {} are displayed as-is.
3. Only double quotes (") are allowed for strings.
4. The statement must end with a semicolon (;).
5. Input values automatically match the variable's declared data type (int, float, str, etc.).
6. Prompt messages must be enclosed in double quotes.

e) Function Definition Statements

Functions in our programming language are reusable blocks of code designed to perform a specific task. Each function must have a name, parameters (optional), and a body enclosed in curly braces {}. Functions may optionally return a value. PyC supports nested functions. All helper functions must be defined inside the main() function or within other blocks. Global functions outside of main() are not supported.

1. Function

Syntax:

```
main(){
# Helper function defined inside main
<data_type> <function_name>(<parameters>){
  <statements>
  [return <expression>;]
}}
```

Sample without return:

```
main(){
void greet(){
  output("Hello, User!");
}
}
```

Sample with return:

```
main(){
int add(int a, int b){
  int c = a + b;
  return c;
}}
```

2. Main Function

The main() function serves as the entry point of every program. Execution starts from the first line inside main() and ends when the closing brace } is reached.

Sample:

```
main(){
  str name = input("Enter your name: ");
  output("Welcome {name}!");
}
```

3. Function Call

Functions are executed by calling their name followed by parentheses containing any required arguments.

Sample:

```
int result = add(10, 20);
output("The sum is {result}");
```

Rules:

1. The function name must start with a letter and cannot be a keyword or reserved word.
2. Parentheses () are required, even if there are no parameters.
3. Parameters are separated by commas.
4. The return statement is optional. If omitted, the function does not return a value.
5. Every function body must be enclosed in curly braces {}.
6. Semicolons (;) are required at the end of each statement inside a function.
7. The arguments provided during a function call must match the parameters in number and data type.

III. Syntax

1. Start Production Rule

<program> → "main" "(" ")" <block>

2. EBNF

<program> → "main" "(" ")" <block>
<block> → "{" { <statement> } "}" | "begin" { <statement> } "end"
<statement> → <declaration_stmt>
| <assignment_stmt>
| <input_stmt>
| <output_stmt>
| <if_stmt>
| <for_stmt>
| <while_stmt>
| <function_def>
| <function_call> ";"
| <array_assignment>
| "break" ";"
| "continue" ";"
| "return" [<expression>] ";"
<declaration_stmt> → <type> <id_list> ";"
<type> → "int" | "float" | "bool" | "char" | "str" | "void"
<id_list> → <identifier> ["[" <expression> "]"] ["=" (<expression> | <array_initializer>)] { "," <identifier> ["[" <expression> "]"] ["=" (<expression> | <array_initializer>)] }
<assignment_stmt> → <identifier> "=" <expression> | <input_call> ";"
<input_stmt> → <identifier> "=" <input_call> ";"
<input_call> → "input" "(" <string_literal> ")" ";" | "input" "(" ")" ";"
<array_assignment> → <identifier> "[" [<expression>] "]" "=" (<array_initializer> | <expression>) ";"
<array_initializer> → "{" [<expression> { "," <expression> }] "}"
<output_stmt> → "output" "(" <string_literal> ")" ";"
<if_stmt> → "if" "(" <expression> ")" ["then"] (<block> | <statement>) { "else" ["if" "(" <expression> ")" ["then"] (<block> | <statement>)] }
<for_stmt> → "for" "(" (<declaration_no_semi> | <assignment_no_semi> | <expression> | "") ";" <expression> ";" <for_update> ")" ["do"] (<block> | <statement>)
<declaration_no_semi> → <type> <id_list_no_semi>
<id_list_no_semi> → <identifier> ["=" <expression>] { "," <identifier> ["=" <expression>] }

<assignment_no_semi> → <identifier> "=" <expression>
<for_update> → <assignment_no_semi> | <identifier> ("++" | "--") | ("++" | "--") <identifier> | <expression>
<while_stmt> → "while" "(" <expression> ")" ["do"] (<block> | <statement>)
<function_def> → <type> <identifier> "(" [<param_list>] ")" <block>
<param_list> → <type> <identifier> { "," <type> <identifier> }
<function_call> → <identifier> "(" [<arg_list>] ")"
<arg_list> → <expression> { "," <expression> }
<expression> → <logical_or>
<logical_or> → <logical_and> { "or" <logical_and> }
<logical_and> → <logical_not> { "and" <logical_not> }
<logical_not> → ["not"] <relational>
<relational> → <arithmetic> [("==" | "!=" | ">" | "<" | ">=" | "<=") <arithmetic>]
<arithmetic> → <term> { ("+" | "-") <term> }
<term> → <factor> { ("*" | "/" | "%" | "//") <factor> }
<factor> → ["+" | "-"] (<number> | <identifier> | <string_literal> | <function_call> | "(" <expression> ")" | <increment_decrement> | <boolean_literal>) ["**" <factor>]
<increment_decrement> → <identifier> "++" | <identifier> "--" | "++" <identifier> | "--" <identifier>
<identifier> → IDENTIFIER_TOKEN
<number> → NUMBER_TOKEN
<string_literal> → STRING_TOKEN | STRING_INTERP_TOKEN
<boolean_literal> → "True" | "False"

3. Example for Each Rules

a) Program

Rule:
<program> → "main" "(" ")" <block>
Example:
```
main() {
  int x;
  output("Hello");
}
```

b) Block

Rule:
<block> → "{" { <statement> } "}"
  | "begin" { <statement> } "end"
Example:
```
begin {
  int x;
  x = 5;
} end
```

c) Declaration Statement

Rule:

<declaration> → <type> <id_list> ";"

Example:

1. Simple Declaration
   `int x;`
2. Simple Declaration with initialization
   `int x = 10;`
3. Multiple Identifiers
   `float a, b, c;`
4. Declaration with initialization
   `int x = 10, y = 5;`

d) Assignment Statement

Rule:

<assignment> → <identifier> "=" (<expression> | <input_call>) ";"

Example:

1. Variable-to-variable
   `a = b;`
2. Constant-to-variable
   `x = 100;`
3. Expression assignment
   `total = price + tax;`
4. Assignment with input
   `name = input("Enter name");`

e) Input Statement

Rule:

<input_stmt> → <identifier> "=" <input_call> ";"

Examples:

1. Valid declaration + input:
   `str name = input("Enter your name: ");`
2. Valid input with empty prompt:
   `name = input();`

f) Output Statement

Rule:

<output_stmt> → "output" "(" <string_literal> ")" ";"

Examples:

1. Normal string
```
output("Hello");
```
2. One variable placeholder
```
output("Value = {x}");
```

g) If Statement

Rule:

<if_stmt> → "if" "(" <expression> ")" ["then"] (<block> | <statement>)
{ "else" ["if" "(" <expression> ")" ["then"] (<block> | <statement>)] }

Examples:

1. Simple IF
```
if (a > b){ output({a}); }
```
2. IF–ELSE
```
if (score >= 75){
  output("Pass");
} else {
  output("Fail"); }
```
3. IF–ELSE IF–ELSE
```
if (x == 0){
  output("Zero");
}else if (x > 0){
  output("Positive");
} else {
  output("Negative");}
```

h) While Statement

Rule:

<while_stmt> → "while" "(" <expression> ")" ["do"] (<block> | <statement>)

Examples:

1. Simple while
```
while (x < 10){ x = x + 1; }
```
2. Nested While
```
while (outer < 3) do {
  int inner = 0;
    while (inner < 3) do {
      output("Outer: {outer}, Inner: {inner}");
    inner = inner + 1;
  }
  outer = outer + 1;
}
```

i) For Statement

Rule: <for_stmt> → "for" "(" (<declaration_no_semi> | <assignment_no_semi> | <expression> | "")
    ";" <expression> ";" <for_update> ")"
    ["do"] (<block> | <statement>)

Examples:

1. Simple for

```
for (int i = 0; i < 5; i++) {
  output("{i}");
}
```

2. Nested

```
for (int i = 0; i < 5; i++) {
        for(int j = 0; j < 5; j++)
  output("{i} and {j}");
}
```

4. Derivation

a.) Declaration Statement

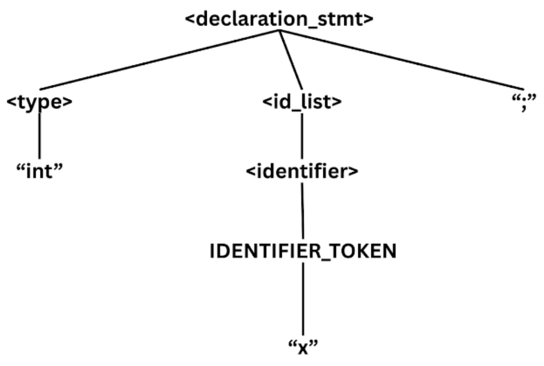
Example 1:

int x;

Leftmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ "int" <id_list> ";"
⇒ "int" <identifier> ";"
⇒ "int" IDENTIFIER_TOKEN ";"
⇒ "int" x ";"

Rightmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ <type> <identifier> ";"
⇒ <type> IDENTIFIER_TOKEN ";"
⇒ <type> x ";"
⇒ "int" x ";"

Parse Tree:

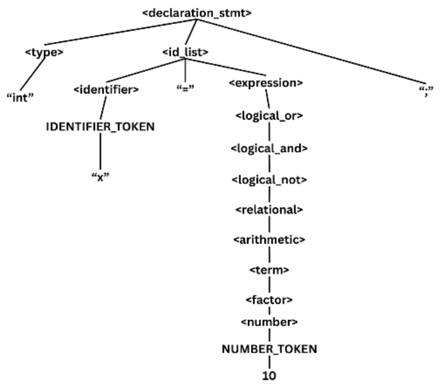


Example 2:
int x = 10;

Leftmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ "int" <id_list> ";"
⇒ "int" <identifier> "=" <expression> ";"
⇒ "int" IDENTIFIER_TOKEN "=" <expression> ";"
⇒ "int" x "=" <expression> ";"
⇒ "int" x "=" <logical_or> ";"
⇒ "int" x "=" <logical_and> ";"
⇒ "int" x "=" <logical_not> ";"
⇒ "int" x "=" <relational> ";"
⇒ "int" x "=" <arithmetic> ";"
⇒ "int" x "=" <term> ";"
⇒ "int" x "=" <factor> ";"
⇒ "int" x "=" <number> ";"
⇒ "int" x "=" NUMBER_TOKEN ";"
⇒ "int" x "=" 10 ";"

Rightmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ <type> <identifier> "=" <expression> ";"
⇒ <type> <identifier> "=" <logical_or> ";"
⇒ <type> <identifier> "=" <logical_and> ";"
⇒ <type> <identifier> "=" <logical_not> ";"
⇒ <type> <identifier> "=" <relational> ";"
⇒ <type> <identifier> "=" <arithmetic> ";"
⇒ <type> <identifier> "=" <term> ";"
⇒ <type> <identifier> "=" <factor> ";"
⇒ <type> <identifier> "=" <number> ";"
⇒ <type> <identifier> "=" NUMBER_TOKEN ";"
⇒ <type> <identifier> "=" 10 ";"
⇒ <type> IDENTIFIER_TOKEN "=" 10 ";"
⇒ <type> x "=" 10 ";"
⇒ "int" x "=" 10 ";"

Parse Tree:

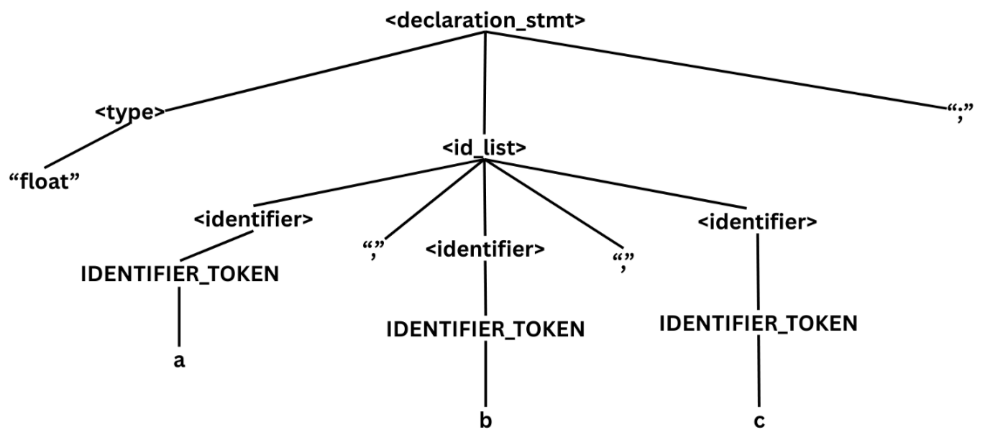


Example 3:
float a, b, c;

Leftmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ "float" <id_list> ";"
⇒ "float" <identifier> "," <identifier> "," <identifier> ";"
⇒ "float" IDENTIFIER_TOKEN "," <identifier> "," <identifier> ";"
⇒ "float" a "," <identifier> "," <identifier> ";"
⇒ "float" a "," IDENTIFIER_TOKEN "," <identifier> ";"
⇒ "float" a "," b "," <identifier> ";"
⇒ "float" a "," b "," IDENTIFIER_TOKEN ";"
⇒ "float" a "," b "," c ";"

Rightmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ <type> <identifier> "," <identifier> "," <identifier> ";"
⇒ <type> <identifier> "," <identifier> "," IDENTIFIER_TOKEN ";"
⇒ <type> <identifier> "," <identifier> "," c ";"
⇒ <type> <identifier> "," IDENTIFIER_TOKEN "," c ";"
⇒ <type> <identifier> "," b "," c ";"
⇒ <type> IDENTIFIER_TOKEN "," b "," c ";"
⇒ <type> a "," b "," c ";"
⇒ "float" a "," b "," c ";"

Parse Tree:



Example 4:
int x = 10, y = 5;

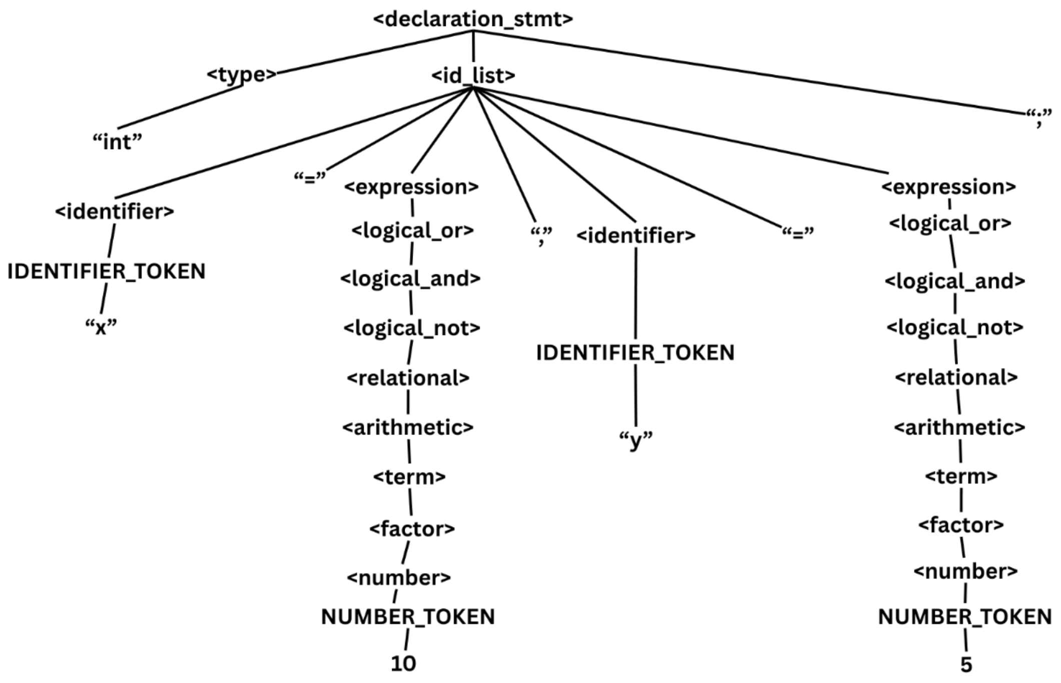
Leftmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ "int" <id_list> ";"
⇒ "int" <identifier> "=" <expression> "," <identifier> "=" <expression> ";"
⇒ "int" IDENTIFIER_TOKEN "=" <expression> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <expression> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <logical_or> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <logical_and> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <logical_not> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <relational> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <arithmetic> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <term> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" <factor> "," <identifier> "=" <expression> ";"

Rightmost Derivation:
⇒ <declaration_stmt>
⇒ <type> <id_list> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <expression> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <logical_or> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <logical_and> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <logical_not> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <relational> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <arithmetic> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <term> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <factor> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" <number> ";"
⇒ <type> <identifier> "=" <expression> "," <identifier> "=" NUMBER_TOKEN ";"

⇒ "int" x "=" <number> "," <identifier> "=" <expression> ";"
⇒ "int" x "=" NUMBER_TOKEN "," <identifier> "=" <expression> ";"
⇒ "int" x "=" 10 "," <identifier> "=" <expression> ";"
⇒ "int" x "=" 10 "," IDENTIFIER_TOKEN "=" <expression> ";"
⇒ "int" x "=" 10 "," y "=" <expression> ";"
⇒ "int" x "=" 10 "," y "=" <logical_or> ";"
⇒ "int" x "=" 10 "," y "=" <logical_and> ";"
⇒ "int" x "=" 10 "," y "=" <logical_not> ";"
⇒ "int" x "=" 10 "," y "=" <relational> ";"
⇒ "int" x "=" 10 "," y "=" <arithmetic> ";"
⇒ "int" x "=" 10 "," y "=" <term> ";"
⇒ "int" x "=" 10 "," y "=" <factor> ";"
⇒ "int" x "=" 10 "," y "=" <number> ";"
⇒ "int" x "=" 10 "," y "=" NUMBER_TOKEN ";"
⇒ "int" x "=" 10 "," y "=" 5 ";"

⇒ <type> <identifier> "=" <expression> "," <identifier> "=" 5 ";"
⇒ <type> <identifier> "=" <expression> "," IDENTIFIER_TOKEN "=" 5 ";"
⇒ <type> <identifier> "=" <expression> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <logical_or> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <logical_and> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <logical_not> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <relational> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <arithmetic> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <term> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <factor> "," y "=" 5 ";"
⇒ <type> <identifier> "=" <number> "," y "=" 5 ";"
⇒ <type> <identifier> "=" NUMBER_TOKEN "," y "=" 5 ";"
⇒ <type> <identifier> "=" 10 "," y "=" 5 ";"
⇒ <type> IDENTIFIER_TOKEN "=" 10 "," y "=" 5 ";"
⇒ <type> x "=" 10 "," y "=" 5 ";"
⇒ "int" x "=" 10 "," y "=" 5 ";"

Parse Tree:



b.) Assignment Statement

Example 1:

a = b;

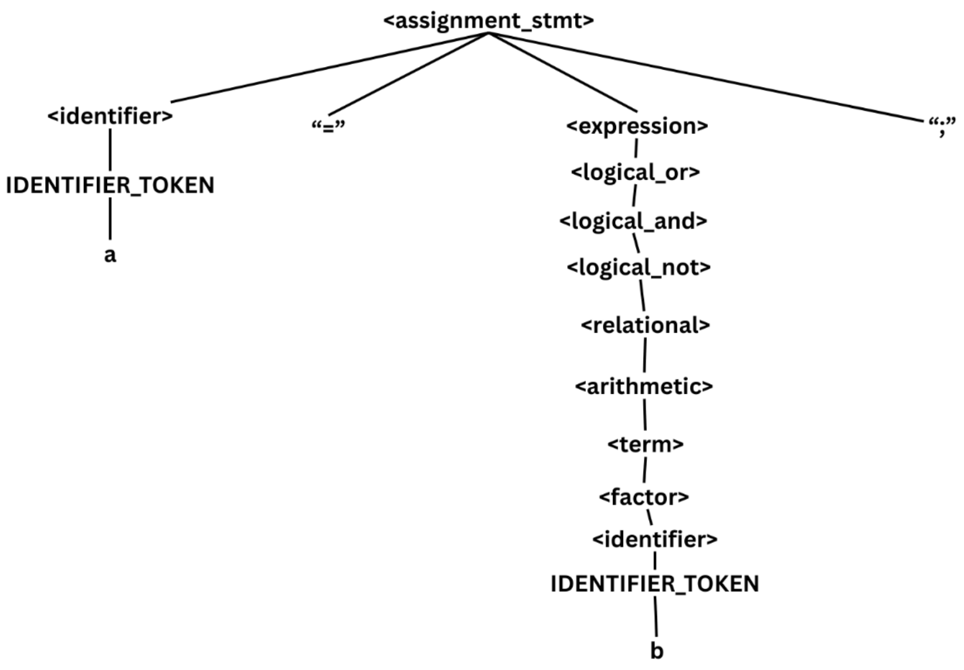
Leftmost Derivation:

⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ IDENTIFIER_TOKEN "=" <expression> ";"
⇒ a "=" <expression> ";"
⇒ a "=" <logical_or> ";"
⇒ a "=" <logical_and> ";"
⇒ a "=" <logical_not> ";"
⇒ a "=" <relational> ";"
⇒ a "=" <arithmetic> ";"
⇒ a "=" <term> ";"
⇒ a "=" <factor> ";"
⇒ a "=" <identifier> ";"
⇒ a "=" IDENTIFIER_TOKEN ";"
⇒ a "=" b ";"

Rightmost Derivation:

⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ <identifier> "=" <logical_or> ";"
⇒ <identifier> "=" <logical_and> ";"
⇒ <identifier> "=" <logical_not> ";"
⇒ <identifier> "=" <relational> ";"
⇒ <identifier> "=" <arithmetic> ";"
⇒ <identifier> "=" <term> ";"
⇒ <identifier> "=" <factor> ";"
⇒ <identifier> "=" <identifier> ";"
⇒ <identifier> "=" IDENTIFIER_TOKEN ";"
⇒ <identifier> "=" b ";"
⇒ IDENTIFIER_TOKEN "=" b ";"
⇒ a "=" b ";"

Parse Tree:

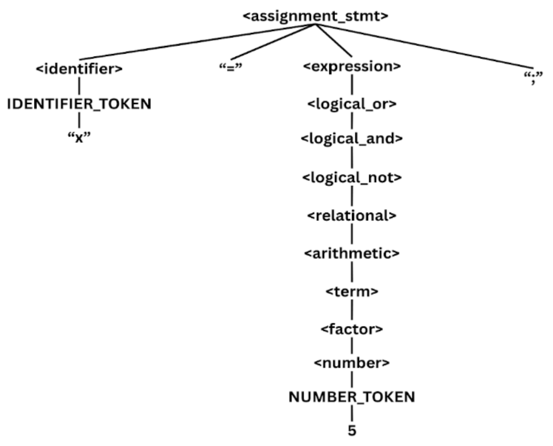


Example 2:
x = 5;

Leftmost Derivation:
⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ IDENTIFIER_TOKEN "=" <expression> ";"
⇒ x "=" <expression> ";"
⇒ x "=" <logical_or> ";"
⇒ x "=" <logical_and> ";"
⇒ x "=" <logical_not> ";"
⇒ x "=" <relational> ";"
⇒ x "=" <arithmetic> ";"
⇒ x "=" <term> ";"
⇒ x "=" <factor> ";"
⇒ x "=" <number> ";"
⇒ x "=" NUMBER_TOKEN ";"
⇒ x "=" 5 ";"

Rightmost Derivation:
⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ <identifier> "=" <logical_or> ";"
⇒ <identifier> "=" <logical_and> ";"
⇒ <identifier> "=" <logical_not> ";"
⇒ <identifier> "=" <relational> ";"
⇒ <identifier> "=" <arithmetic> ";"
⇒ <identifier> "=" <term> ";"
⇒ <identifier> "=" <factor> ";"
⇒ <identifier> "=" <number> ";"
⇒ <identifier> "=" NUMBER_TOKEN ";"
⇒ <identifier> "=" 100 ";"
⇒ IDENTIFIER_TOKEN "=" 100 ";"
⇒ x "=" 5 ";"

Parse Tree:

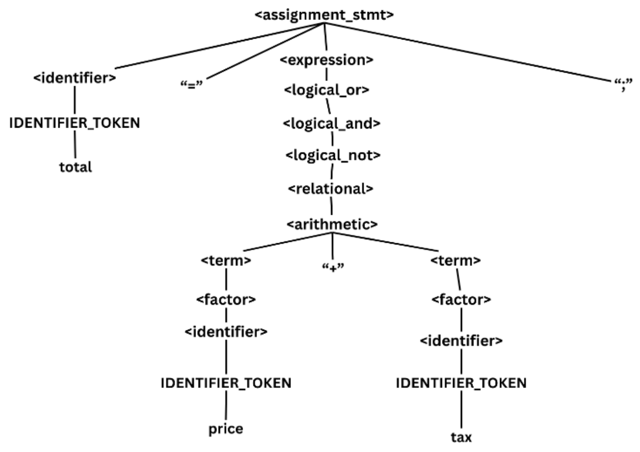


Example 3:
total = price + tax;

Leftmost Derivation:
⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ IDENTIFIER_TOKEN "=" <expression> ";"
⇒ total "=" <expression> ";"
⇒ total "=" <logical_or> ";"
⇒ total "=" <logical_and> ";"
⇒ total "=" <logical_not> ";"
⇒ total "=" <relational> ";"
⇒ total "=" <arithmetic> ";"
⇒ total "=" <term> "+" <term> ";"
⇒ total "=" <factor> "+" <term> ";"
⇒ total "=" <identifier> "+" <term> ";"
⇒ total "=" IDENTIFIER_TOKEN "+" <term> ";"
⇒ total "=" price "+" <term> ";"
⇒ total "=" price "+" <factor> ";"
⇒ total "=" price "+" <identifier> ";"
⇒ total "=" price "+" IDENTIFIER_TOKEN ";"
⇒ total "=" price "+" tax ";"

Rightmost Derivation:
⇒ <assignment_stmt>
⇒ <identifier> "=" <expression> ";"
⇒ <identifier> "=" <logical_or> ";"
⇒ <identifier> "=" <logical_and> ";"
⇒ <identifier> "=" <logical_not> ";"
⇒ <identifier> "=" <relational> ";"
⇒ <identifier> "=" <arithmetic> ";"
⇒ <identifier> "=" <term> "+" <term> ";"
⇒ <identifier> "=" <term> "+" <factor> ";"
⇒ <identifier> "=" <term> "+" <identifier> ";"
⇒ <identifier> "=" <term> "+" IDENTIFIER_TOKEN ";"
⇒ <identifier> "=" <term> "+" tax ";"
⇒ <identifier> "=" <factor> "+" tax ";"
⇒ <identifier> "=" <identifier> "+" tax ";"
⇒ <identifier> "=" IDENTIFIER_TOKEN "+" tax ";"
⇒ <identifier> "=" price "+" tax ";"
⇒ IDENTIFIER_TOKEN "=" price "+" tax ";"
⇒ total "=" price "+" tax ";"

Parse Tree:

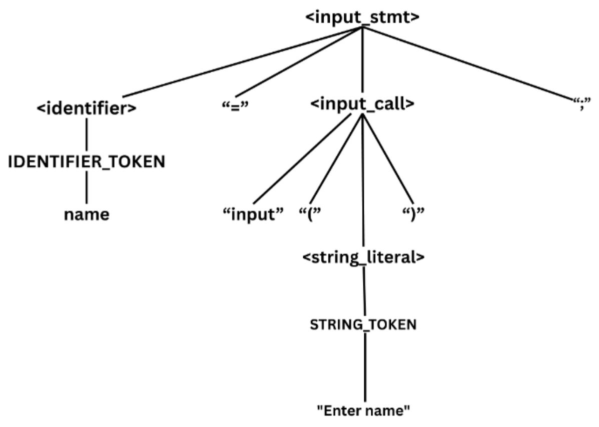


Example 4:
name = input("Enter name");

Leftmost Derivation:
⇒ <input_stmt>
⇒ <identifier> "=" <input_call> ";"
⇒ IDENTIFIER_TOKEN "=" <input_call> ";"
⇒ name "=" <input_call> ";"
⇒ name "=" "input" "(" <string_literal> ")" ";"
⇒ name "=" "input" "(" STRING_TOKEN ")" ";"
⇒ name "=" "input" "(" "Enter name" ")" ";"

Rightmost Derivation:
⇒ <input_stmt>
⇒ <identifier> "=" <input_call> ";"
⇒ <identifier> "=" "input" "(" <string_literal> ")" ";"
⇒ <identifier> "=" "input" "(" STRING_TOKEN ")" ";"
⇒ <identifier> "=" "input" "(" "Enter name" ")" ";"
⇒ IDENTIFIER_TOKEN "=" "input" "(" "Enter name" ")" ";"
⇒ name "=" "input" "(" "Enter name" ")" ";"

Parse Tree:



c.) Input Statement

Example 1:

str name = input("Enter your name: ");

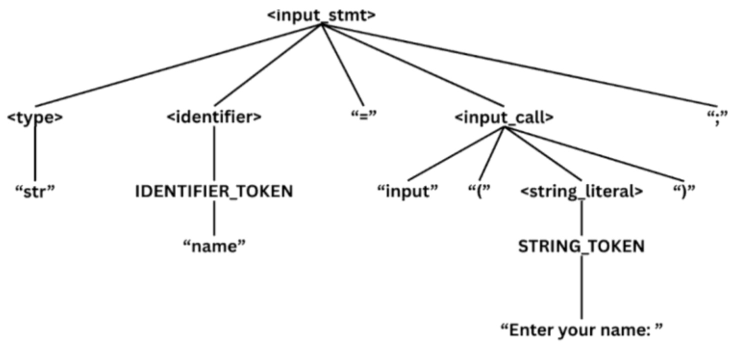
Leftmost Derivation:

<input_stmt>
⇒ <type> <identifier> "=" <input_call> ";"
⇒ "str" <identifier> "=" <input_call> ";"
⇒ "str" IDENTIFIER_TOKEN "=" <input_call> ";"
⇒ "str" IDENTIFIER_TOKEN "=" "input" "(" <string_literal> ")" ";"
⇒ "str" IDENTIFIER_TOKEN "=" "input" "(" STRING_TOKEN ")" ";"
⇒ str name = input("Enter your name: ");

Rightmost Derivation:

<input_stmt>
⇒ <type> <identifier> "=" <input_call> ";"
⇒ <type> <identifier> "=" "input" "(" <string_literal> ")" ";"
⇒ <type> <identifier> "=" "input" "(" STRING_TOKEN ")" ";"
⇒ <type> IDENTIFIER_TOKEN "=" "input" "(" STRING_TOKEN ")" ";"
⇒ "str" IDENTIFIER_TOKEN "=" "input" "(" STRING_TOKEN ")" ";"
⇒ str name = input("Enter your name: ");

Parse Tree:



Example 2:

name = input();

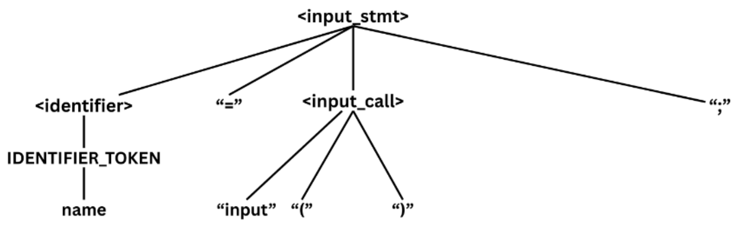
Leftmost Derivation:

⇒ <input_stmt>
⇒ <identifier> "=" <input_call> ";"
⇒ IDENTIFIER_TOKEN "=" <input_call> ";"
⇒ name "=" <input_call> ";"
⇒ name "=" "input" "(" ")" ";"

Rightmost Derivation:

⇒ <input_stmt>
⇒ <identifier> "=" <input_call> ";"
⇒ <identifier> "=" "input" "(" ")" ";"
⇒ IDENTIFIER_TOKEN "=" "input" "(" ")" ";"
⇒ name "=" "input" "(" ")" ";"

Parse Tree:



d.) Output Statement

Example 1:

output("Hello");

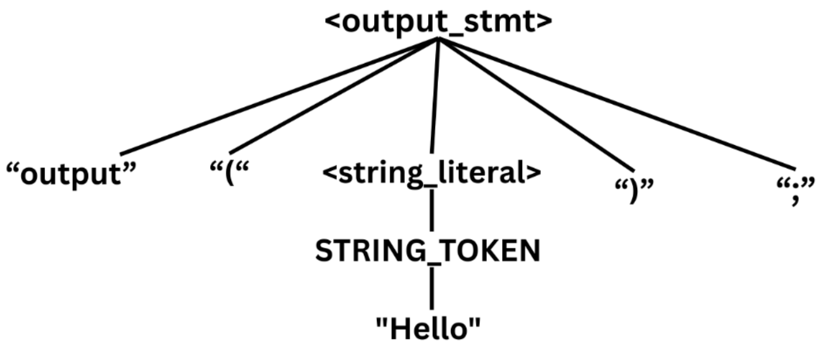
Leftmost Derivation:

⇒ <output_stmt>
⇒ "output" "(" <string_literal> ")" ";"
⇒ "output" "(" STRING_TOKEN ")" ";"
⇒ "output" "(" "Hello" ")" ";"

Rightmost Derivation:

⇒ <output_stmt>
⇒ "output" "(" <string_literal> ")" ";"
⇒ "output" "(" STRING_TOKEN ")" ";"
⇒ "output" "(" "Hello" ")" ";"

Parse Tree:



Example 2:

output("Value = {x}");

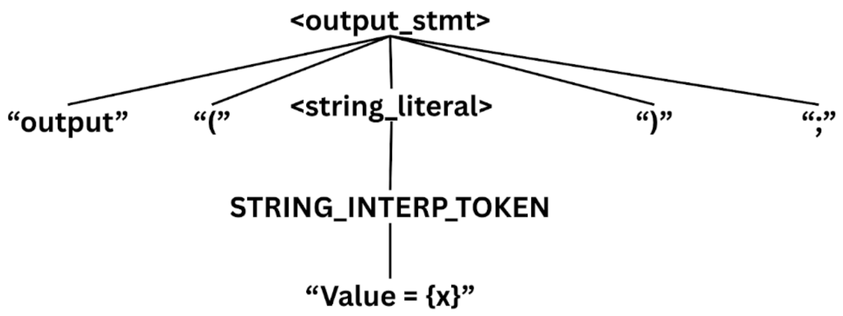
Leftmost Derivation:

<output_stmt>
⇒ "output" "("<string_literal>")" ";"
⇒ "output" "(" STRING_INTERP_TOKEN ")" ";"
⇒ output("Value = {x}");

Rightmost Derivation:

<output_stmt>
⇒ "output" "("<string_literal>")" ";"
⇒ "output" "(" STRING_INTERP_TOKEN ")" ";"
⇒ output("Value = {x}");

Parse Tree:



e.) Conditional Statement

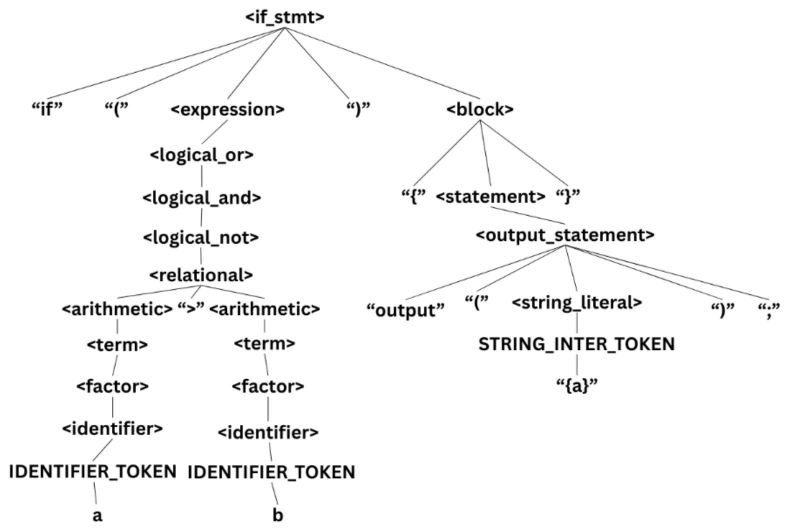
Example 1:

if (a > b){ output("{a}"); }

Leftmost Derivation:
⇒ <if_stmt>
⇒ "if" "(" <expression> ")" <block>
⇒ "if" "(" <logical_or> ")" <block>
⇒ "if" "(" <logical_and> ")" <block>
⇒ "if" "(" <logical_not> ")" <block>
⇒ "if" "(" <relational> ")" <block>
⇒ "if" "(" <arithmetic> ">" <arithmetic> ")" <block>
⇒ "if" "(" <term> ">" <arithmetic> ")" <block>
⇒ "if" "(" <factor> ">" <arithmetic> ")" <block>
⇒ "if" "(" <identifier> ">" <arithmetic> ")" <block>
⇒ "if" "(" IDENTIFIER_TOKEN ">" <arithmetic> ")" <block>
⇒ "if" "(" a ">" <arithmetic> ")" <block>
⇒ "if" "(" a ">" <term> ")" <block>
⇒ "if" "(" a ">" <factor> ")" <block>
⇒ "if" "(" a ">" <identifier> ")" <block>
⇒ "if" "(" a ">" IDENTIFIER_TOKEN ")" <block>
⇒ "if" "(" a ">" b ")" <block>
⇒ "if" "(" a ">" b ")" "{" <statement> "}"
⇒ "if" "(" a ">" b ")" "{" <output_stmt> "}"
⇒ "if" "(" a ">" b ")" "{" "output" "(" <string_literal> ")" ";" "}"
⇒ "if" "(" a ">" b ")" "{" "output" "(" STRING_INTERP_TOKEN ")" ";" "}"
⇒ if (a > b) {output("{a}");}

Rightmost Derivation:
⇒ <if_stmt>
⇒ "if" "(" <expression> ")" <block>
⇒ "if" "(" <expression> ")" "{" <statement> "}"
⇒ "if" "(" <expression> ")" "{" <output_stmt> "}"
⇒ "if" "(" <expression> ")" "{" "output" "(" <string_literal> ")" ";" "}"
⇒ "if" "(" <expression> ")" "{" "output" "(" STRING_INTERP_TOKEN ")" ";" "}"
⇒ "if" "(" <expression> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <logical_or> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <logical_and> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <logical_not> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <relational> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <arithmetic> ">" <arithmetic> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <arithmetic> ">" <factor> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <arithmetic> ">" <identifier> ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <arithmetic> ">" IDENTIFIER_TOKEN ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <arithmetic> ">" b ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <factor> ">" b ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" <identifier> ">" b ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ "if" "(" IDENTIFIER_TOKEN ">" b ")" "{" "output" "(" "{a}" ")" ";" "}"
⇒ if (a > b) {output("{a}"); }

Parse Tree:



Example 2:
if (score >= 75) then {
output("Pass");
} else {
output("Fail");
}

Leftmost Derivation:
<if_stmt>
⇒ "if" "(" <expression> ")" <block>
⇒ "if" "(" <relational> ")" <block>
⇒ "if" "(" <arithmetic> ">=" <arithmetic> ")" <block>
⇒ "if" "(" <term> ">=" <arithmetic> ")" <block>
⇒ "if" "(" <factor> ">=" <arithmetic> ")" <block>
⇒ "if" "(" <identifier> ">=" <arithmetic> ")" <block>
⇒ "if" "(" IDENTIFIER_TOKEN ">=" <arithmetic> ")" <block>
⇒ "if" "(" score ">=" <arithmetic> ")" <block>
⇒ "if" "(" score ">=" <term> ")" <block>
⇒ "if" "(" score ">=" <factor> ")" <block>
⇒ "if" "(" score ">=" <number> ")" <block>

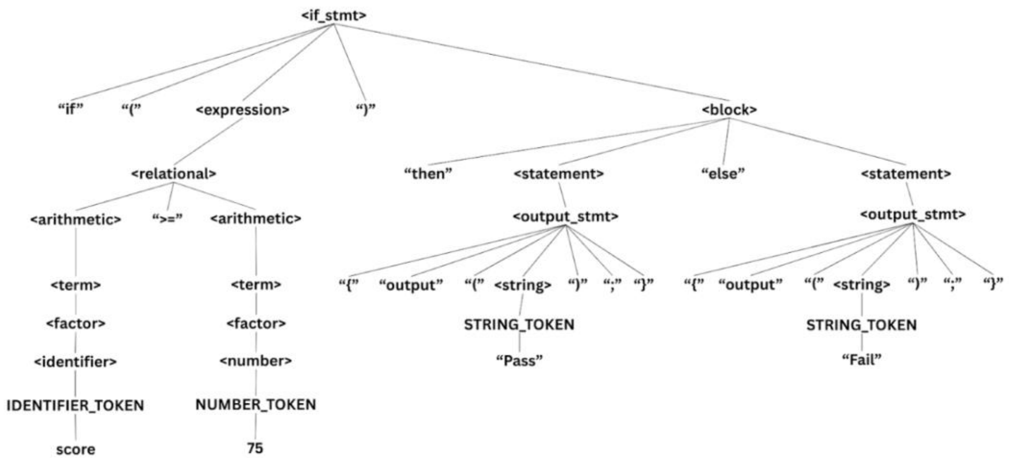
Rightmost Derivation:
<if_stmt>
⇒ "if" "(" <expression> ")" <block>
⇒ "if" "(" <expression> ")" "then" <statement> "else" <statement>
⇒ "if" "(" <expression> ")" "then" <statement> "else" <output_stmt>
⇒ "if" "(" <expression> ")" "then" <statement> "else" "{" "output" "(" <string_literal> ")" "}" ";"
⇒ "if" "(" <expression> ")" "then" <statement> "else" "{" "output" "(" STRING_TOKEN ")" "}" ";"
⇒ "if" "(" <expression> ")" "then" <statement> "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <expression> ")" "then" <output_stmt>"else" "{" "output" "(" Fail")" "}" ";"

⇒ "if" "(" score ">=" NUMBER_TOKEN ")" <block>
⇒ "if" "(" score ">=" 75 ")" <block>
⇒ "if" "(" score ">=" 75 ")" "then" <statement> "else" <statement>
⇒ "if" "(" score ">=" 75 ")" "then" <output_stmt> "else" <statement>
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" <string_literal> ")" ";" "}" "else" <statement>
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" STRING_TOKEN ")" ";" "}" "else" <statement>
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" <statement>
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" <output_stmt>
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" <string_literal> ")" "}" ";"
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" STRING_TOKEN ")" "}" ";"
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ if (score >= 75) then {
output("Pass");
} else {
output("Fail");
}

⇒ "if" "(" <expression> ")" "then" "{" "output" "(" <string_literal> ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <expression> ")" "then" "{" "output" "(" STRING_TOKEN ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <expression> ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <relational> ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" <arithmetic> ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" <term> ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" <factor>")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" <number> ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" NUMBER_TOKEN ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <arithmetic> ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <term> ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <factor> ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" <identifier> ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"

⇒ "if" "(" IDENTIFIER_TOKEN ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ "if" "(" score ">=" 75 ")" "then" "{" "output" "(" Pass ")" ";" "}" "else" "{" "output" "(" Fail")" "}" ";"
⇒ if (score >= 75) then {
output("Pass");
} else {
output("Fail");
}

Parse Tree:



Example 3:

```
if (x == 0) {
   output("Zero");
} else if (x > 0) {
   output("Positive");
} else {
   output("Negative");
}
```

Leftmost Derivation:
⇒ <if_stmt>
⇒ "if" "(" <expression> ")" <block> { "else" ... }
⇒ "if" "(" <logical_or> ")" <block> { "else" ... }
⇒ "if" "(" <logical_and> ")" <block> { "else" ... }

Rightmost Derivation:
⇒ <if_stmt>
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" <block>

⇒ "if" "(" <logical_not> ")" <block> { "else" ... }
⇒ "if" "(" <relational> ")" <block> { "else" ... }
⇒ "if" "(" <arithmetic> "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" <term> "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" <factor> "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" <identifier> "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" IDENTIFIER_TOKEN "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" <term> ")" <block> { "else" ... }
⇒ "if" "(" x "==" <factor> ")" <block> { "else" ... }
⇒ "if" "(" x "==" <number> ")" <block> { "else" ... }
⇒ "if" "(" x "==" NUMBER_TOKEN ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" <statement> "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" <output_stmt> "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" <string_literal> ")" ";" "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" STRING_TOKEN ")" ";" "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <expression> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <logical_or> ")" <block> { "else" ... }
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" <statement> "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" <output_stmt> "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" "output" "(" <string_literal> ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" "output" "(" STRING_TOKEN ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" <block> "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" <statement> "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" <output_stmt> "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" "output" "(" <string_literal> ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" "output" "(" STRING_TOKEN ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <expression> ")" "{" "output" "(" "Positive"

⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <logical_and> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <logical_not> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <relational> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <arithmetic> ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <term> ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <factor> ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" <identifier> ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" IDENTIFIER_TOKEN ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" <arithmetic> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" <term> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" <factor> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" <number> ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" NUMBER_TOKEN ")" <block> { "else" ... }
")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <logical_or> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <logical_and> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <logical_not> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <relational> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" <arithmetic> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" <term> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" <factor> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" <number> ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" NUMBER_TOKEN ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"

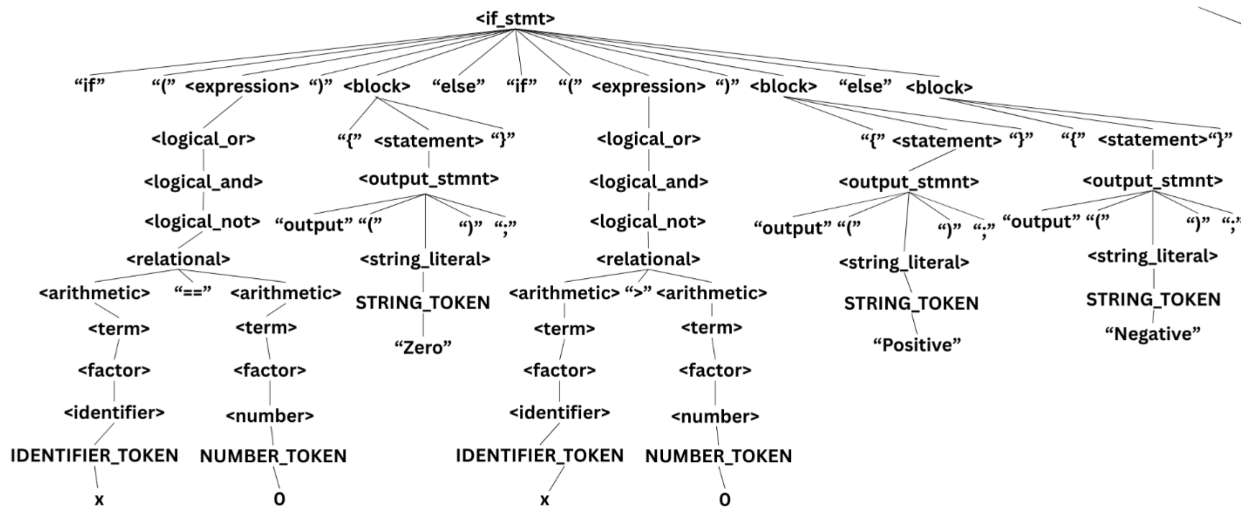
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" <block> { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" <statement> "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" <output_stmt> "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" <string_literal> ")" ";" "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" STRING_TOKEN ")" ";" "}" { "else" ... }
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" { "else" ... } *(Expanding the second iteration of the else group - the final ELSE)*
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" <block>
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" <statement> "}"
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" <output_stmt> "}"
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" <string_literal> ")" ";" "}"
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" STRING_TOKEN ")" ";" "}"
⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <arithmetic> ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <term> ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <factor> ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" <identifier> ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" IDENTIFIER_TOKEN ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" <block> "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" "{" <statement> "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <expression> ")" "{" <output_stmt> "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"

"Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"	⇒ "if" "(" <expression> ")" "{" "output" "(" <string_literal> ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <expression> ")" "{" "output" "(" STRING_TOKEN ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <expression> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <expression> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <logical_or> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <logical_and> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <logical_not> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <relational> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <arithmetic> "==" <arithmetic> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}" ⇒ "if" "(" <arithmetic> "==" <term> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">"

0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <arithmetic> "==" <factor> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <arithmetic> "==" <number> ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <arithmetic> "==" NUMBER_TOKEN ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <arithmetic> "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <arithmetic> "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <term> "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <factor> "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" <identifier> "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"
⇒ "if" "(" IDENTIFIER_TOKEN "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"

⇒ "if" "(" x "==" 0 ")" "{" "output" "(" "Zero" ")" ";" "}" "else" "if" "(" x ">" 0 ")" "{" "output" "(" "Positive" ")" ";" "}" "else" "{" "output" "(" "Negative" ")" ";" "}"

Parse Tree:



f.) Iterative Statement
Example (While Loop):
while (x < 10){ x = x + 1; }

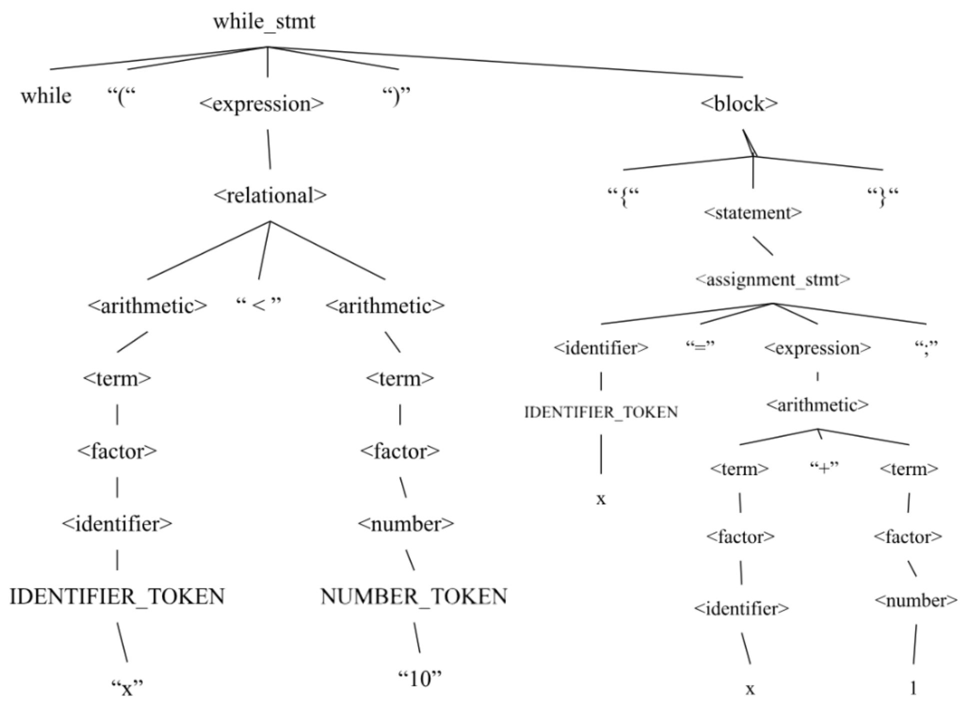
Leftmost Derivation:
<while_stmt>
⇒ "while" "(" <expression> ")" <block>
⇒ "while" "(" <relational> ")" <block>
⇒ "while" "(" <arithmetic> "<" <arithmetic> ")" <block>
⇒ "while" "(" <term> "<" <arithmetic> ")" <block>
⇒ "while" "(" <factor> "<" <arithmetic> ")" <block>
⇒ "while" "(" <identifier> "<" <arithmetic> ")" <block>
⇒ "while" "(" IDENTIFIER_TOKEN "<" <arithmetic> ")" <block>
⇒ "while" "(" x "<" <arithmetic> ")" <block>
⇒ "while" "(" x "<" <term> ")" <block>
⇒ "while" "(" x "<" <factor> ")" <block>

Rightmost Derivation:
<while_stmt>
⇒ "while" "(" <expression> ")" <block>
⇒ "while" "(" <relational> ")" <block>
⇒ "while" "(" <arithmetic> "<" <arithmetic> ")" <block>
⇒ "while" "(" <arithmetic> "<" <term> ")" <block>
⇒ "while" "(" <arithmetic> "<" <factor> ")" <block>
⇒ "while" "(" <arithmetic> "<" <number> ")" <block>
⇒ "while" "(" <arithmetic> "<" NUMBER_TOKEN ")" <block>
⇒ "while" "(" <arithmetic> "<" 10 ")" <block>
⇒ "while" "(" <term> "<" 10 ")" <block>
⇒ "while" "(" <factor> "<" 10 ")" <block>

⇒ "while" "(" x "<" <number> ")" <block>
⇒ "while" "(" x "<" NUMBER_TOKEN ")" <block>
⇒ "while" "(" x "<" 10 ")" <block>
⇒ "while" "(" x "<" 10 ")" "{" <statement> "}"
⇒ "while" "(" x "<" 10 ")" "{" <assignment_stmt> "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <expression> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" IDENTIFIER_TOKEN "=" <expression> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" <expression> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" <arithmetic> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" <term> "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" <factor> "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" <identifier> "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" IDENTIFIER_TOKEN "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" x "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" x "+" <factor> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" x "+" <number> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" x "+" NUMBER_TOKEN ";" "}"
⇒ while (x < 10){ x = x + 1; }

⇒ "while" "(" <identifier> "<" 10 ")" <block>
⇒ "while" "(" IDENTIFIER_TOKEN "<" 10 ")" <block>
⇒ "while" "(" x "<" 10 ")" <block>
⇒ "while" "(" x "<" 10 ")" "{" <statement> "}"
⇒ "while" "(" x "<" 10 ")" "{" <assignment_stmt> "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <expression> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <arithmetic> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <term> "+" <term> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <term> "+" <factor> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <term> "+" <number> ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <term> "+" NUMBER_TOKEN ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <factor> "+" 1 ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" <identifier> "+" 1 ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" IDENTIFIER_TOKEN "+" 1 ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" <identifier> "=" x "+" 1 ";" "}"
⇒ "while" "(" x "<" 10 ")" "{" x "=" x "+" 1 ";" "}"
⇒ while (x < 10){ x = x + 1; }

Parse Tree:



Example (For Loop):
for (i = 0; i < 5; i++){ i = i + 1; }

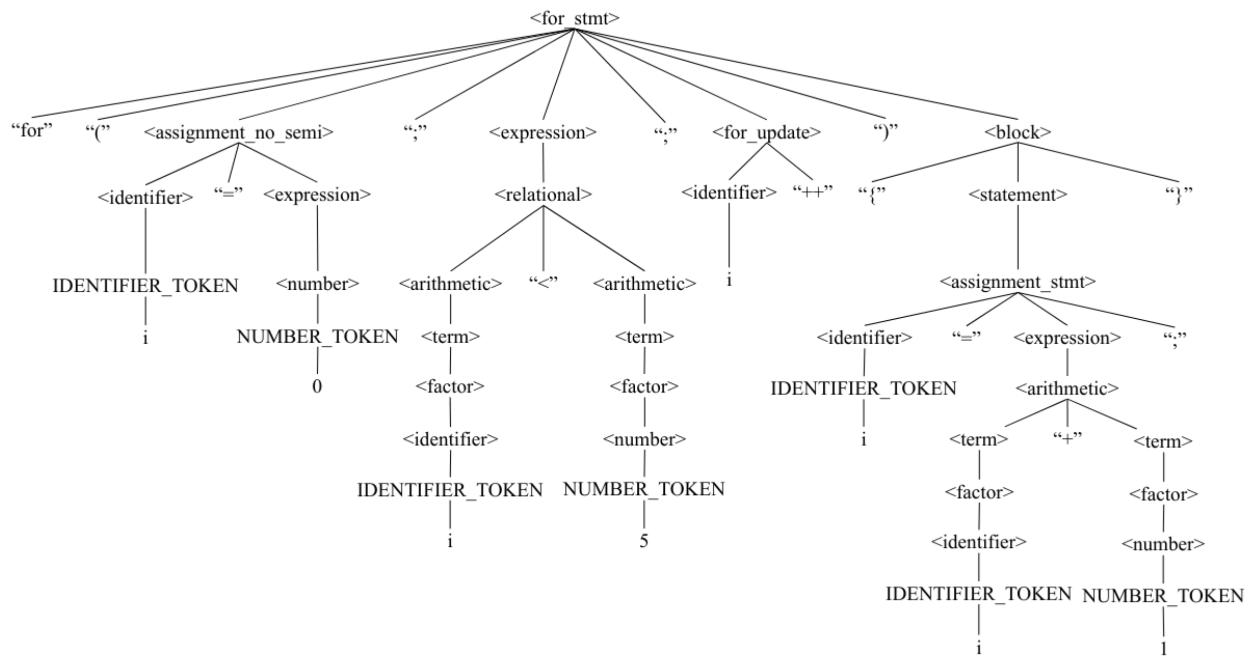
Leftmost Derivation:
<for_stmt>
⇒ "for" "(" <assignment_no_semi> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" <identifier> "=" <expression> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" IDENTIFIER_TOKEN "=" <expression> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" i "=" <expression> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" i "=" <number> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" i "=" NUMBER_TOKEN ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" i "=" 0 ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" i "=" 0 ";" <relational> ";" <for_update> ")" <block>

Rightmost Derivation:
<for_stmt>
⇒ "for" "(" <assignment_no_semi> ";" <expression> ";" <for_update> ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <expression> ";" <identifier> "++" ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <expression> ";" IDENTIFIER_TOKEN "++" ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <expression> ";" i "++" ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <relational> ";" i "++" ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <arithmetic> "<" <arithmetic> ";" i "++" ")" <block>
⇒ "for" "(" <assignment_no_semi> ";" <identifier> "<" <number> ";" i "++" ")" <block>

⇒ “for” “(” i “=” 0 “;” <arithmetic> “<” <arithmetic> “;” <for_update> “)” <block>
⇒ “for” “(” i “=” 0 “;” <identifier> “<” <number> “;” <for_update> “)” <block>
⇒ “for” “(” i “=” 0 “;” IDENTIFIER_TOKEN “<” NUMBER_TOKEN “;” <for_update> “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” <for_update> “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” <identifier> “++” “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” IDENTIFIER_TOKEN “++” “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ <statement> “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ <assignment_stmt> “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ <identifier> “=” <expression> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” <arithmetic> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” <term> “+” <term> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” <identifier> “+” <number> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” i “+” 1 “;” “}”
⇒ for (i = 0; i < 5; i++){ i = i + 1; }

⇒ “for” “(” <assignment_no_semi> “;” IDENTIFIER_TOKEN “<” NUMBER_TOKEN “;” i “++” “)” <block>
⇒ “for” “(” <assignment_no_semi> “;” i “<” 5 “;” i “++” “)” <block>
⇒ “for” “(” <identifier> “=” <number> “;” i “<” 5 “;” i “++” “)” <block>
⇒ “for” “(” IDENTIFIER_TOKEN “=” NUMBER_TOKEN “;” i “<” 5 “;” i “++” “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” <block>
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ <statement> “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ <assignment_stmt> “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” <expression> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” <arithmetic> “;” “}”
⇒ “for” “(” i “=” 0 “;” i “<” 5 “;” i “++” “)” “{“ i “=” i “+” 1 “;” “}”
⇒ for (i = 0; i < 5; i++){ i = i + 1; }

Parse Tree:


<for_stmt>
"for"
"("
<assignment_no_semi>
";"
<expression>
";"
<for_update>
")"
<block>
<identifier>
"="
<expression>
<relational>
<identifier>
"++"
"{"
<statement>
"}"
IDENTIFIER_TOKEN
i
<number>
NUMBER_TOKEN
0
<arithmetic>
"<"
<arithmetic>
<term>
<factor>
<identifier>
IDENTIFIER_TOKEN
i
<term>
<factor>
<number>
NUMBER_TOKEN
5
i
<assignment_stmt>
<identifier>
"="
<expression>
";"
IDENTIFIER_TOKEN
i
<arithmetic>
<term>
"+"
<term>
<factor>
<identifier>
IDENTIFIER_TOKEN
i
<factor>
<number>
NUMBER_TOKEN
1