Al-Azhar University

Faculty of Engineering

Systems and Computer Engineering Department

Mini Math Compiler

Analysis Phases Report

SCE 410 / Compilers

Prepared by:

Mostafa Khaled Mohamed Hassanin (73)
Mohamed Ramadan Saeed Mahmoud (56)
Ahmed Hosny Abdel Hameed (7)
Mahmoud Mohamed Ahmed Hussein (70)
Mohamed Shaaban Abdel Maqsoud El-Shenawy (57)

Supervised by:

DR Ahmed Abd Al Aziz

December 2025

Contents

1 Introduction

1.1 Project Overview

The Mini Math Compiler is a compiler designed to process mathematical expressions. It transforms human-readable mathematical notation into structured representations that can be analyzed and evaluated.

1.2 Scope

This compiler focuses exclusively on the analysis phases of compilation:

- **Lexical Analysis**: Converting source text into a stream of tokens
- **Syntax Analysis**: Building an Abstract Syntax Tree (AST) from tokens
- **Semantic Analysis**: Performing type inference and building a symbol table

1.3 Supported Features

The Mini Math Compiler supports the following language features:

Feature	Description	Examples
Integers	Whole numbers without decimal points	`42`, `0`, `123`
Floats	Decimal numbers with fractional parts	`3.14`, `0.5`, `2.0`
Variables	Named storage for values	`x`, `result`, `total`
Arithmetic Operators	Basic mathematical operations	`+`, `-`, `*`, `/`, `^`
Assignment	Storing values in variables	`x = 42`, `y = x + 1`
Parentheses	Grouping expressions	`(2 + 3) * 4`
Unary Operators	Sign operators	`-5`, `+3`

2 Lexical Analysis

2.1 What is Lexical Analysis?

Lexical analysis is the first phase of compilation. It transforms raw source code (a stream of characters) into a sequence of meaningful units called **tokens**. This process is also known as **tokenization** or **scanning**.

The lexer reads the source text character by character, grouping characters into lexemes and classifying each lexeme with a token type. For example, the characters 4, 2 are grouped into the lexeme 42 and classified as an INTEGER token.

The lexer also tracks the position (line and column) of each token in the source code, which is essential for error reporting. When the lexer encounters an unexpected character, it produces an ERROR token rather than halting, allowing the compiler to report multiple errors in a single pass.

2.2 The Scanning Process

The Mini Math Compiler's lexer performs **character-by-character scanning** with position tracking. The scanning algorithm works as follows:

1. **Initialize** the scanner at position (line 1, column 1)
2. **Read** the next character from the source
3. **Classify** the character:
    - If it's whitespace (space, tab, carriage return), skip it
    - If it's a newline, increment the line counter and reset the column
    - If it's a digit, scan a complete number (INTEGER or FLOAT)
    - If it's a letter or underscore, scan an identifier
    - If it's an operator or punctuation, create the corresponding token
    - Otherwise, create an ERROR token
4. **Record** the token with its lexeme and position
5. **Repeat** until end of input
6. **Append** an EOF token to mark the end

2.3 Lexer Algorithm (Pseudocode)

The following pseudocode illustrates the main tokenization algorithm:

```
function tokenize(source):
  tokens = []
  current = 0
  line = 1
  column = 1

  while current < length(source):
    start = current
    startColumn = column
    char = source[current]

    # Single-character tokens
    if char is '+', '-', '*', '/', '^', '(', ')', '=':
      tokens.add(Token(type=charToTokenType(char),
                       lexeme=char,
                       position=(line, startColumn)))
      current++
      column++

    # Skip whitespace
    else if char is ' ', '\t', '\r':
      current++
      column++

    # Handle newlines
    else if char is '\n':
      line++
      column = 1
      current++
```

```
    # Scan numbers
    else if isDigit(char):
      (token, newCurrent) = scanNumber(source, current, line, startColumn)
      tokens.add(token)
      current = newCurrent
      column += (newCurrent - start)

    # Scan identifiers
    else if isAlpha(char):
      (token, newCurrent) = scanIdentifier(source, current, line,
↪   startColumn)
      tokens.add(token)
      current = newCurrent
      column += (newCurrent - start)

    # Unexpected character
    else:
      tokens.add(Token(type=ERROR,
                       lexeme=char,
                       position=(line, startColumn)))
      current++
      column++

  # Add EOF token
  tokens.add(Token(type=EOF, lexeme="", position=(line, column)))
  return tokens


function scanNumber(source, start, line, column):
  current = start
  isFloat = false
```

```
  # Scan integer part
  while current < length(source) and isDigit(source[current]):
    current++

  # Check for decimal point
  if source[current] is '.' and isDigit(source[current+1]):
    isFloat = true
    current++  # consume '.'

    # Scan fractional part
    while current < length(source) and isDigit(source[current]):
      current++

  lexeme = source[start:current]
  value = parseFloat(lexeme)
  tokenType = FLOAT if isFloat else INTEGER

  token = Token(type=tokenType,
                lexeme=lexeme,
                literal=value,
                position=(line, column))

  return (token, current)


function scanIdentifier(source, start, line, column):
  current = start

  # Scan alphanumeric characters
  while current < length(source) and isAlphaNumeric(source[current]):
    current++
```

```
  lexeme = source[start:current]
  token = Token(type=IDENTIFIER,
                lexeme=lexeme,
                position=(line, column))

  return (token, current)
```

2.4 Token Structure

Each token produced by the lexer contains the following information:

```
interface Token {
  type: TokenType;     // The category of the token
  lexeme: string;      // The actual text from source code
  position: Position; // Location in source (line, column)
  literal?: number;    // Numeric value (for INTEGER and FLOAT only)
}

interface Position {
  line: number;        // Line number (starting from 1)
  column: number;      // Column number (starting from 1)
}
```

Field	Description	Example
`type`	Token category (one of 13 types)	`INTEGER`, `PLUS`, `IDENTIFIER`
`lexeme`	The exact text matched from source	`"42"`, `"+"`, `"result"`
`position`	Source location for error reporting	`{ line: 1, column: 5 }`
`literal`	Parsed numeric value (optional)	`42`, `3.14`

The `literal` field is only present for `INTEGER` and `FLOAT` tokens, storing the parsed numeric value for direct use by later compiler phases.

2.5 Token Types

The Mini Math Compiler recognizes 13 token types:

Type	Description	Example Lexeme	Has Literal
`INTEGER`	Whole numbers without decimal point	`42`, `0`, `123`	Yes
`FLOAT`	Numbers with decimal point	`3.14`, `0.5`, `2.0`	Yes
`IDENTIFIEI`	Variable names (letters, digits, underscore)	`x`, `result`, `var_1`	No
`PLUS`	Addition operator	`+`	No
`MINUS`	Subtraction operator	`-`	No
`STAR`	Multiplication operator	`*`	No
`SLASH`	Division operator	`/`	No
`CARET`	Exponentiation operator	`^`	No
`LPAREN`	Left parenthesis	`(`	No
`RPAREN`	Right parenthesis	`)`	No
`EQUALS`	Assignment operator	`=`	No
`EOF`	End of file marker	(empty)	No
`ERROR`	Invalid/unexpected character	`@`, `#`, `$`	No

2.6 Number Recognition

The lexer distinguishes between INTEGER and FLOAT tokens based on the presence of a decimal point. The number scanning algorithm:

1. **Scan integer part**: Consume all consecutive digits
2. **Check for decimal**: Look ahead for a . followed by a digit
3. **Scan fractional part**: If decimal found, consume the . and all following digits
4. **Classify**: Token is FLOAT if decimal was found, otherwise INTEGER

```
Input: "42"     → INTEGER (no decimal point)
Input: "3.14"   → FLOAT   (has decimal point)
Input: "2.0"    → FLOAT   (has decimal point)
Input: "100"    → INTEGER (no decimal point)
```

The lexer uses **lookahead** to ensure the decimal point is followed by a digit. This prevents 42. from being incorrectly parsed—the . would be treated as an unexpected character if not followed by digits.

```
// Simplified number scanning logic
if (peek() === '.' && isDigit(peekNext())) {
  isFloat = true;
  advance(); // consume '.'
  while (isDigit(peek())) advance();
}
```

2.7 Error Handling

When the lexer encounters an unexpected character, it produces an ERROR token instead of halting. This approach allows the compiler to continue scanning and report multiple errors in a single compilation pass.

Characters that produce ERROR tokens include: - Special characters not in the language: @, #, $, &, !, etc. - Unicode characters outside ASCII letters and digits - Any character not recognized as part of a valid token

```
// Error token creation
```

```
{
  type: 'ERROR',
  lexeme: '@',            // The unexpected character
  position: { line: 1, column: 5 }
}
```

The lexer records the position of each error token, enabling precise error messages that point to the exact location of the problem in the source code.

2.8 Tokenization Example

Input Source Code:

```
x = 42 + 3.14
```

Output Token Stream:

#	Type	Lexeme	Position	Literal
1	IDENTIFIER	x	line 1, col 1	—
2	EQUALS	=	line 1, col 3	—
3	INTEGER	42	line 1, col 5	42
4	PLUS	+	line 1, col 8	—
5	FLOAT	3.14	line 1, col 10	3.14
6	EOF	(empty)	line 1, col 14	—

Note how whitespace characters (spaces between tokens) are consumed but do not produce tokens. The lexer tracks column positions accurately, accounting for multi-character lexemes like 42 and 3.14.

3 Syntax Analysis

3.1 What is Syntax Analysis?

Syntax analysis is the second phase of compilation. It takes the stream of tokens produced by the lexer and constructs an **Abstract Syntax Tree (AST)**—a hierarchical representation of the program's structure. This process is also known as **parsing**.

While lexical analysis answers "what are the words?", syntax analysis answers "how do the words fit together?". The parser verifies that tokens appear in a valid order according to the language's grammar and builds a tree structure that captures the relationships between expressions.

For example, given the tokens for `2 + 3 * 4`, the parser must recognize that multiplication has higher precedence than addition, producing a tree where `3 * 4` is computed first.

3.2 Recursive Descent Parsing

The Mini Math Compiler uses **recursive descent parsing**, a top-down parsing technique where each grammar rule is implemented as a function. The parser starts from the highest-level rule (program) and recursively descends through the grammar to parse sub-expressions.

Key characteristics: - **Top-down**: Starts from the root of the parse tree and works down - **Predictive**: Uses lookahead to decide which production to apply - **Direct mapping**: Each grammar rule corresponds to a parsing function

The parser also employs **precedence climbing** to handle operator precedence correctly. Each precedence level has its own parsing function, with lower-precedence operators calling higher-precedence functions for their operands.

3.3 Grammar Specification

The Mini Math Compiler's expression language is defined by the following grammar in BNF (Backus-Naur Form) notation:

BNF Grammar Specification

program	→	*statement* *	*(zero or more)*
statement	→	*assignment* | *expression*	
assignment	→	IDENTIFIER "=" *expression*	
expression	→	*additive*	
additive	→	*multiplicative* (("+" | "-") *multiplicative*)*	
multiplicative	→	*power* (("*" | "/") *power*)*	
power	→	*unary* ("^" *power*)?	*(right-associative)*
unary	→	("-" | "+") *unary* | *primary*	
primary	→	INTEGER | FLOAT | IDENTIFIER | "(" *expression* ")"	

Figure 1: BNF Grammar Specification

Grammar Notation: - * means zero or more repetitions - ? means zero or one (optional) - | means alternative choices - Quoted strings are literal tokens - UPPERCASE names are terminal tokens from the lexer

3.4 Parser Algorithm (Pseudocode)

The following pseudocode illustrates the recursive descent parser with precedence climbing:

```
function parse(tokens):
```

```
  current = 0
  statements = []
  errors = []

  while tokens[current].type != EOF:
    try:
      statement = parseStatement()
      statements.add(statement)
    catch error:
      errors.add(error)
      break

  return (statements, errors)


function parseStatement():
  # Check for assignment: IDENTIFIER = expression
  if tokens[current].type is IDENTIFIER and tokens[current+1].type is
   ↪  EQUALS:
    return parseAssignment()
  else:
    return parseExpression()


function parseAssignment():
  nameToken = advance()  # consume IDENTIFIER
  advance()              # consume EQUALS
  value = parseExpression()

  return AssignmentNode(name=nameToken.lexeme,
                        value=value,
                        position=nameToken.position)
```

```
# Expression parsing with precedence climbing
function parseExpression():
  return parseAdditive()


# Precedence Level 1: Addition and Subtraction (left-associative)
function parseAdditive():
  left = parseMultiplicative()

  while tokens[current].type is PLUS or MINUS:
    operator = advance().lexeme
    right = parseMultiplicative()
    left = BinaryExprNode(operator, left, right)

  return left


# Precedence Level 2: Multiplication and Division (left-associative)
function parseMultiplicative():
  left = parsePower()

  while tokens[current].type is STAR or SLASH:
    operator = advance().lexeme
    right = parsePower()
    left = BinaryExprNode(operator, left, right)

  return left


# Precedence Level 3: Exponentiation (right-associative)
```

```
function parsePower():
  left = parseUnary()

  if tokens[current].type is CARET:
    operator = advance().lexeme
    # Right-associative: recursively call parsePower
    right = parsePower()
    return BinaryExprNode(operator, left, right)

  return left


# Precedence Level 4: Unary operators
function parseUnary():
  if tokens[current].type is MINUS or PLUS:
    operator = advance().lexeme
    operand = parseUnary()  # Allow chained unary operators
    return UnaryExprNode(operator, operand)

  return parsePrimary()


# Primary expressions (highest precedence)
function parsePrimary():
  # Integer literal
  if tokens[current].type is INTEGER:
    token = advance()
    return LiteralNode(value=token.literal, dataType=Integer)

  # Float literal
  if tokens[current].type is FLOAT:
    token = advance()
```

```
    return LiteralNode(value=token.literal, dataType=Float)

  # Variable reference
  if tokens[current].type is IDENTIFIER:
    token = advance()
    return VariableNode(name=token.lexeme)

  # Parenthesized expression
  if tokens[current].type is LPAREN:
    advance()  # consume '('
    expr = parseExpression()
    expect(RPAREN)  # consume ')'
    return expr

  error("Expected expression")


function advance():
  token = tokens[current]
  current++
  return token
```

3.5 Operator Precedence and Associativity

Operators are parsed according to their precedence level. Higher precedence operators bind more tightly than lower precedence operators.

Level	Operators	Description	Associativity
1 (lowest)	+, -	Addition, Subtraction	Left-to-right
2	*, /	Multiplication, Division	Left-to-right
3	^	Exponentiation	Right-to-left

Level	Operators	Description	Associativity
4 (highest)	unary -, +	Negation, Positive	Right-to-left

Associativity Examples:

Left-associative (evaluated left to right):

$$8 - 4 - 2 \rightarrow (8 - 4) - 2 \rightarrow 2$$

$$12 \div 3 \div 2 \rightarrow (12 \div 3) \div 2 \rightarrow 2$$

Right-associative (evaluated right to left):

$$2^{3^2} \rightarrow 2^{(3^2)} \rightarrow 2^9 \rightarrow 512$$

$$-(-5) \rightarrow 5$$

Precedence Example:

$$2 + 3 \times 4^2$$

Parsed as: $2 + (3 \times (4^2)) = 2 + (3 \times 16) = 2 + 48 = 50$

3.6 AST Node Types

The parser produces five types of AST nodes:

3.6.1 1. Assignment Node

Represents variable assignment: `IDENTIFIER = expression`

```
interface AssignmentNode {
  kind: 'Assignment';
  name: string;         // Variable name
  value: ASTNode;       // Expression being assigned
  position: Position;   // Source location
}
```

Example: `x = 42` produces:

Field	Value
kind	Assignment
name	x
value	Literal(42, Integer)

3.6.2 2. Binary Expression Node

Represents operations with two operands: `left operator right`

```
interface BinaryExprNode {
  kind: 'BinaryExpr';
  operator: '+' | '-' | '*' | '/' | '^';
  left: ASTNode;        // Left operand
  right: ASTNode;       // Right operand
  position: Position;
}
```

Example: `3 + 4` produces:

Field	Value
kind	BinaryExpr
operator	+

Field	Value
left	Literal(3)
right	Literal(4)

3.6.3 3. Unary Expression Node

Represents operations with one operand: `operator operand`

```
interface UnaryExprNode {
  kind: 'UnaryExpr';
  operator: '-' | '+';
  operand: ASTNode;    // The operand
  position: Position;
}
```

Example: `-5` produces:

Field	Value
kind	UnaryExpr
operator	-
operand	Literal(5)

3.6.4 4. Literal Node

Represents numeric values (integers or floats):

```
interface LiteralNode {
  kind: 'Literal';
  value: number;       // Numeric value
  dataType: 'Integer' | 'Float';
  position: Position;
```

```
}
```

Examples: - 42 → { kind: "Literal", value: 42, dataType: "Integer" } - 3.14 → { kind: "Literal", value: 3.14, dataType: "Float" }

3.6.5 5. Variable Node

Represents a variable reference:

```
interface VariableNode {
  kind: 'Variable';
  name: string;        // Variable name
  position: Position;
}
```

Example: x produces:

```
| Field | Value |
|-------|-------|
| kind | Variable |
| name | x |


## Parsing Example

**Input Expression:**
```typescript
2 + 3 * 4
```

Parsing Process:

1. parseExpression() calls parseAdditive()
2. parseAdditive() calls parseMultiplicative() for left operand
3. parseMultiplicative() calls parsePower() → parseUnary() →
 ↪ parsePrimary()
4. parsePrimary() matches INTEGER token 2, returns Literal(2)
```

```
5. Back in parseAdditive(): matches +, calls parseMultiplicative() for
 ↪  right
6. parseMultiplicative() parses 3, matches *, parses 4
7. parseMultiplicative() returns BinaryExpr(*, 3, 4)
8. parseAdditive() returns BinaryExpr(+, 2, BinaryExpr(*, 3, 4))
```

Resulting AST:

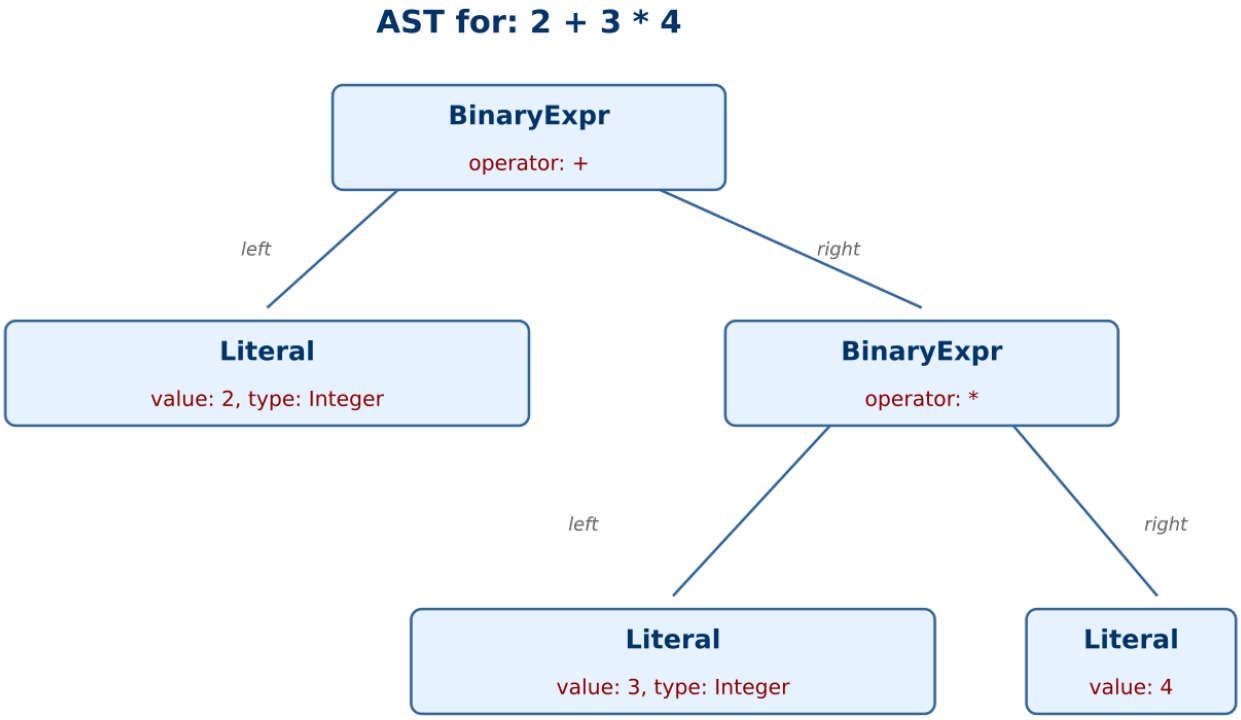

Figure: Parsing Example - Abstract Syntax Tree

Figure 2: Parsing Example AST

AST Structure:

Node	Field	Value
Root	kind	BinaryExpr
Root	operator	+
Root	left	Literal(2, Integer)
Root	right	BinaryExpr (see below)
Right child	kind	BinaryExpr
Right child	operator	*
Right child	left	Literal(3, Integer)
Right child	right	Literal(4, Integer)

The tree structure correctly captures that 3 * 4 should be evaluated before adding 2, respecting operator precedence.

4 Semantic Analysis

4.1 What is Semantic Analysis?

Semantic analysis is the third phase of compilation, following lexical and syntax analysis. While the parser verifies that code is grammatically correct, the semantic analyzer ensures the code is **meaningful**—that operations make sense and variables are properly defined.

The Mini Math Compiler's semantic analyzer performs three key tasks:

1. **Type Inference**: Determines the data type (Integer or Float) of every expression
2. **Symbol Table Construction**: Builds a mapping of variable names to their types and definition locations
3. **Error Detection**: Identifies undefined variable references

The semantic analyzer traverses the AST produced by the parser, annotating each node with its resolved type and building the symbol table as it encounters assignments.

4.2 Semantic Analyzer Algorithm (Pseudocode)

The following pseudocode illustrates the semantic analysis algorithm:

```
function analyze(ast):
  symbolTable = empty Map
  annotatedAst = []
  errors = []

  for each node in ast:
    annotatedNode = analyzeNode(node, symbolTable, errors)
    annotatedAst.add(annotatedNode)

  return (symbolTable, annotatedAst, errors)


function analyzeNode(node, symbolTable, errors):
```

```
  if node.kind is Assignment:
    return analyzeAssignment(node, symbolTable, errors)
  else if node.kind is BinaryExpr:
    return analyzeBinaryExpr(node, symbolTable, errors)
  else if node.kind is UnaryExpr:
    return analyzeUnaryExpr(node, symbolTable, errors)
  else if node.kind is Literal:
    return analyzeLiteral(node)
  else if node.kind is Variable:
    return analyzeVariable(node, symbolTable, errors)


function analyzeAssignment(node, symbolTable, errors):
  # Analyze the value expression
  annotatedValue = analyzeNode(node.value, symbolTable, errors)
  valueType = getResolvedType(annotatedValue)

  # Add variable to symbol table
  if valueType is not null:
    symbolTable[node.name] = SymbolEntry(
      name=node.name,
      type=valueType,
      definedAt=node.position
    )

  return AssignmentNode(name=node.name,
                        value=annotatedValue,
                        position=node.position)


function analyzeBinaryExpr(node, symbolTable, errors):
  # Analyze operands
```

```
  annotatedLeft = analyzeNode(node.left, symbolTable, errors)
  annotatedRight = analyzeNode(node.right, symbolTable, errors)

  leftType = getResolvedType(annotatedLeft)
  rightType = getResolvedType(annotatedRight)

  # Type promotion rules
  if leftType and rightType:
    # Division always produces Float
    if node.operator is '/':
      resolvedType = Float
    # If either operand is Float, result is Float
    else if leftType is Float or rightType is Float:
      resolvedType = Float
    # Both are Integer
    else:
      resolvedType = Integer
  else:
    resolvedType = null

  return BinaryExprNode(operator=node.operator,
                        left=annotatedLeft,
                        right=annotatedRight,
                        resolvedType=resolvedType,
                        position=node.position)


function analyzeUnaryExpr(node, symbolTable, errors):
  # Analyze operand
  annotatedOperand = analyzeNode(node.operand, symbolTable, errors)
  operandType = getResolvedType(annotatedOperand)
```

```
  # Unary operator preserves type
  return UnaryExprNode(operator=node.operator,
                       operand=annotatedOperand,
                       resolvedType=operandType,
                       position=node.position)


function analyzeLiteral(node):
  # Type is determined by dataType field from parser
  return LiteralNode(value=node.value,
                     dataType=node.dataType,
                     resolvedType=node.dataType,
                     position=node.position)


function analyzeVariable(node, symbolTable, errors):
  # Look up variable in symbol table
  entry = symbolTable.get(node.name)

  if entry is null:
    errors.add(Error(
      phase=semantic,
      message="Undefined variable '" + node.name + "'",
      position=node.position,
      variableName=node.name
    ))
    return VariableNode(name=node.name, position=node.position)

  return VariableNode(name=node.name,
                      resolvedType=entry.type,
                      position=node.position)
```

```
function getResolvedType(node):
  if node.kind is Assignment:
    return getResolvedType(node.value)
  else:
    return node.resolvedType
```

4.3 Type System

The Mini Math Compiler supports two data types:

Type	Description	Example Values
`Integer`	Whole numbers without decimal point	`42`, `0`, `-7`
`Float`	Numbers with decimal point	`3.14`, `0.5`, `-2.0`

4.3.1 Type Inference for Literals

The type of a literal is determined during parsing based on its lexical form:

- **No decimal point** → `Integer` (e.g., 42 is Integer)
- **Has decimal point** → `Float` (e.g., `3.14` is Float)

```
// Literal node with inferred type
{
  kind: 'Literal',
  value: 42,
  dataType: 'Integer',  // Set by parser
  resolvedType: 'Integer'  // Confirmed by semantic analyzer
}
```

4.4 Type Promotion Rules

When binary operations combine different types, the semantic analyzer applies **type promotion** rules to determine the result type:

Left Operand	Operator	Right Operand	Result Type
Integer	+, -, *, ^	Integer	Integer
Integer	+, -, *, ^	Float	Float
Float	+, -, *, ^	Integer	Float
Float	+, -, *, ^	Float	Float
Any	/	Any	**Float** (always)

Key Rules:

1. **Mixed operands promote to Float**: If either operand is Float, the result is Float
2. **Division always produces Float**: Even $4 \div 2$ results in Float type
3. **Unary operators preserve type**: $-x$ has the same type as x

$2 + 3 \rightarrow$ Integer (both operands Integer)

$2 + 3.0 \rightarrow$ Float (right operand is Float)

$2.0 + 3 \rightarrow$ Float (left operand is Float)

$4 \div 2 \rightarrow$ Float (division always Float)

$-5 \rightarrow$ Integer (unary preserves type)

$-5.0 \rightarrow$ Float (unary preserves type)

4.5 Symbol Table

The symbol table is a data structure that maps variable names to their type information. It is built incrementally as the semantic analyzer processes assignment statements.

4.5.1 Structure

```
type SymbolTable = Map<string, SymbolEntry>;

interface SymbolEntry {
  name: string;      // Variable name
  type: DataType;    // Inferred type (Integer or Float)
  definedAt: Position; // Source location of definition
}
```

4.5.2 How Variables Are Added

When the analyzer encounters an assignment (`x = expression`):

1. Analyze the right-hand expression to determine its type
2. Create a symbol entry with the variable name and inferred type
3. Store the entry in the symbol table (overwrites if variable exists)

```
// Processing: x = 42
symbolTable.set('x', {
  name: 'x',
  type: 'Integer',
  definedAt: { line: 1, column: 1 }
});
```

4.5.3 How Variables Are Looked Up

When the analyzer encounters a variable reference:

1. Look up the variable name in the symbol table
2. If found, use the stored type as the resolved type
3. If not found, report an “undefined variable” error

4.6 Error Detection

The semantic analyzer detects **undefined variable** errors—when code references a variable that has not been assigned a value.

4.6.1 Undefined Variable Detection

When a variable reference is encountered, the analyzer checks if it exists in the symbol table. If not found, an error is recorded:

```
interface CompilerError {
  phase: 'semantic';
  message: string;        // "Undefined variable 'x'"
  position: Position;     // Location of the reference
  variableName: string;   // The undefined variable name
}
```

4.6.2 Example Error

```
Input: y = x + 1
```

If x has not been previously assigned, the analyzer produces:

```
{
  phase: 'semantic',
  message: "Undefined variable 'x'",
  position: { line: 1, column: 5 },
  variableName: 'x'
}
```

The analyzer continues processing after detecting an error, allowing multiple undefined variable errors to be reported in a single pass.

4.7 Semantic Analysis Example

Input Source Code:

```
x = 10
```

```
y = 3.14
z = x + y
```

4.7.1 Step-by-Step Analysis

Statement 1: `x = 10` - Analyze literal `10` → type is `Integer` - Add to symbol table: `x` → `Integer`

Statement 2: `y = 3.14` - Analyze literal `3.14` → type is `Float` - Add to symbol table: `y` → `Float`

Statement 3: `z = x + y` - Look up x → found, type is `Integer` - Look up y → found, type is `Float` - Apply type promotion: `Integer + Float = Float` - Add to symbol table: `z` → `Float`

4.7.2 Resulting Symbol Table

Variable	Type	Defined At
x	Integer	line 1, col 1
y	Float	line 2, col 1
z	Float	line 3, col 1

4.7.3 Annotated AST (Simplified)

Annotated AST with Resolved Types

```
Assignment (x)
+-- Literal(10, resolvedType: Integer)

Assignment (y)
+-- Literal(3.14, resolvedType: Float)

Assignment (z)
+-- BinaryExpr(+, resolvedType: Float)
    |-- Variable (x, resolvedType: Integer)    Integer + Float
    +-- Variable (y, resolvedType: Float)       = Float
```

Figure 3: Annotated AST with Resolved Types

Each node in the annotated AST includes a `resolvedType` field indicating the inferred type, enabling later compiler phases (such as code generation) to make type-aware decisions.

5 Compilation Pipeline

5.1 Overview

The Mini Math Compiler processes source code through three sequential phases, each transforming the input into a more structured representation. This pipeline architecture allows each phase to focus on a specific aspect of analysis while passing its output to the next phase.

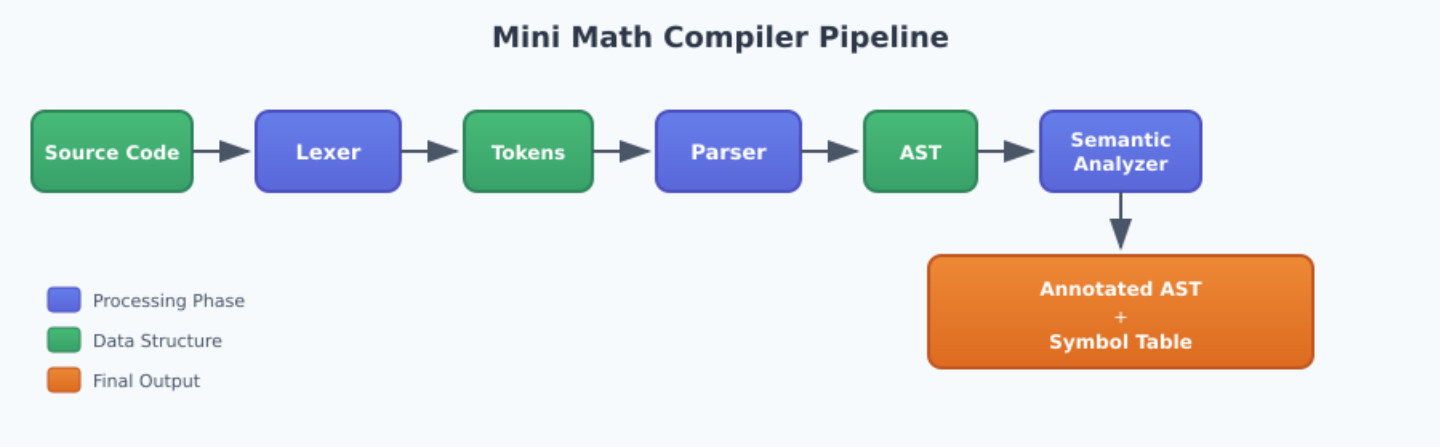

Figure 4: Compilation Pipeline Diagram

5.2 Phase Outputs

5.2.1 Phase 1: Lexical Analysis (Lexer)

Input: Source code as a string of characters

Output: Array of Token objects

```
interface Token {
  type: TokenType;      // INTEGER, FLOAT, IDENTIFIER, PLUS, etc.
  lexeme: string;       // The actual text from source
  position: Position;   // Line and column number
  literal?: number;     // Parsed value for numbers
}
```

Example:

```
Input:  "x = 42 + 3.14"
Output: [IDENTIFIER(x), EQUALS, INTEGER(42), PLUS, FLOAT(3.14), EOF]
```

The lexer transforms unstructured text into a sequence of categorized tokens, discarding whitespace and tracking source positions for error reporting.

5.2.2 Phase 2: Syntax Analysis (Parser)

Input: Array of Token objects from the lexer

Output: Array of AST nodes + parsing errors

```
interface ParseResult {
  ast: ASTNode[];           // Array of statement nodes
  errors: CompilerError[];  // Syntax errors encountered
}
```

AST Node Types: - `Assignment` — Variable assignment statements - `BinaryExpr` — Operations with two operands (+, -, *, /, ^) - `UnaryExpr` — Operations with one operand (-, +) - `Literal` — Numeric values (Integer or Float) - `Variable` — Variable references

Example:

```
Input:  [IDENTIFIER(x), EQUALS, INTEGER(2), PLUS, INTEGER(3)]
Output: Assignment(x, BinaryExpr(+, Literal(2), Literal(3)))
```

The parser verifies grammatical correctness and builds a tree structure that captures operator precedence and expression nesting.

5.2.3 Phase 3: Semantic Analysis

Input: AST nodes from the parser

Output: Annotated AST + Symbol Table + semantic errors

```
interface SemanticResult {
```

```
  ast: ASTNode[];           // AST with resolvedType annotations
  symbolTable: SymbolTable; // Variable name → type mapping
  errors: CompilerError[];  // Semantic errors (undefined variables)
}

type SymbolTable = Map<string, {
  name: string;
  type: 'Integer' | 'Float';
  definedAt: Position;
}>;
```

Example:

```
Input:  Assignment(x, BinaryExpr(+, Literal(2), Literal(3.0)))
Output:
  - Annotated AST with resolvedType: Float (due to type promotion)
  - Symbol Table: { x → Float }
```

The semantic analyzer adds type information to every AST node and builds a symbol table mapping variables to their inferred types.

5.3 Complete Pipeline Example

Source Code:

```
x = 10
y = x + 2.5
```

Phase 1 — Lexer Output:

```
[IDENTIFIER(x), EQUALS, INTEGER(10), EOF]
[IDENTIFIER(y), EQUALS, IDENTIFIER(x), PLUS, FLOAT(2.5), EOF]
```

Phase 2 — Parser Output:

```
Assignment(x, Literal(10, Integer))
Assignment(y, BinaryExpr(+, Variable(x), Literal(2.5, Float)))
```

Phase 3 — Semantic Output:

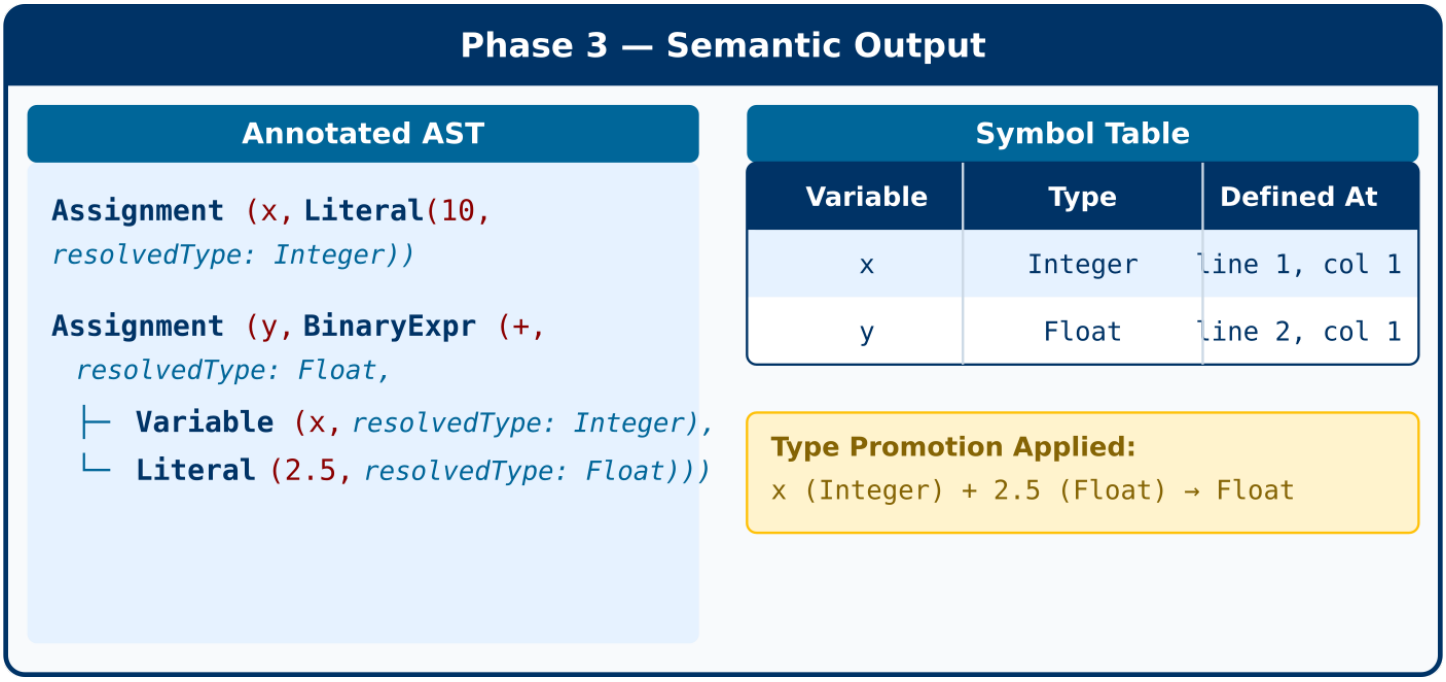

Figure 5: Phase 3 Semantic Output

The final output provides everything needed for code generation: a fully typed AST and a complete symbol table of all defined variables.