Chapter 5: Creating TinyCL - A Complete Programming Language

In this chapter, we'll explore the complete implementation of TinyCL (Tiny C-Like Language), a fully-functional programming language built using our PEG parser library. TinyCL demonstrates how to create a production-quality language with parser, interpreter, and multi-target compiler.

5.1 TinyCL Language Overview

TinyCL is a comprehensive programming language that showcases all aspects of language implementation:

5.1.1 Language Features

TinyCL supports the following modern language features:

1. **Variables and Constants**: `var x = 10;` and `const PI = 3;`
2. **Full Arithmetic**: `+`, `-`, `*`, `/` with proper precedence
3. **Logical Operations**: &&, `||`, `!` for boolean logic
4. **Comparison Operations**: ==, `!`=, <, >, <=, >=
5. **Arrays**: `[1, 2, 3]` and array access `arr[0]`
6. **Characters**: `'A'` for single characters
7. **Control Structures**: If-else statements and while loops
8. **Functions**: Declaration, calls, and return values
9. **Comments**: `# This is a comment`
10. **String Operations**: Concatenation and manipulation

Here's an example of a TinyCL program:

```
```

Calculate factorial with functions and arrays

var numbers = [5, 6, 7]; const MAX = 10;

func factorial(n) { if (n <= 1) { return 1; } else { return n * factorial(n - 1); } }

var i = 0; while (i < 3) { var num = numbers[i]; var result = factorial(num); print("Factorial of " + num + " is " + result); i = i + 1; } ```

5.1.2 Complete Grammar Specification

Here's the complete EBNF grammar for TinyCL:

```ebnf Program ::= Statements

Statements ::= Statement*

Statement ::= FunctionDecl | VariableDecl | ConstantDecl | "if" "(" Expression ")" Block ( "else" Block )? | "while" "(" Expression ")" Block | "print" "(" Expression ")" ";" | "return" Expression? ";" | Id "=" Expression ";" | Id "(" Arguments? ")" ";" | Block | Comment

FunctionDecl ::= "func" Id "(" Parameters? ")" Block VariableDecl ::= "var" Id "=" Expression ";" ConstantDecl ::= "const" Id "=" Expression ";"

Parameters ::= Id ( "," Id ) *Arguments ::= Expression ( "," Expression )*

Block ::= "{" Statements? "}"

# Expression hierarchy with proper precedence

Expression ::= LogicalOr LogicalOr ::= LogicalAnd ( "||" LogicalAnd ) *LogicalAnd ::= Equality ( "&&" Equality )* Equality ::= Comparison ( ( "!=" | "==" ) Comparison ) *Comparison ::= Term ( ( "<=" | ">=" | "<" | ">" ) Term )* Term ::= Factor ( ( "+" | "-" )
```

Factor) *Factor ::= Unary (("*" | "/") Unary) Unary ::= ("!" | "-")? Postfix Postfix ::= Primary ("[" Expression "]")*

Primary ::= "(" Expression ")" | Id "(" Arguments? ")" | "[" Arguments? "]" | Id | Number | String | Character | "true" | "false"

Literals

String ::= '"' StringChar* '"' StringChar ::= [#x20-#x21] | [#x23-#x5B] | [#x5D-#x7E] | "\" EscapeChar EscapeChar ::= '"' | "'" | "\" | "n" | "r" | "t" | "0" | "b" | "f" | "v" | "l"

Character ::= "'" CharChar "'" CharChar ::= [#x20-#x26] | [#x28-#x5B] | [#x5D-#x7E] | "\" EscapeChar

Id ::= Letter (Letter | Digit | "_")* Number ::= Digit+ Letter ::= [a-zA-Z] Digit ::= [0-9]

Comment ::= "#" [^\n]* ```

This comprehensive grammar defines all the syntax of TinyCL programs, including modern features like arrays, logical operators, and proper expression precedence.

5.2 Complete Implementation Overview

The TinyCL implementation consists of three main components:

1. **Parser** (`src/tinycl/parser.py`) - Converts source code to AST
2. **Interpreter** (`src/tinycl/interpreter.py`) - Executes TinyCL programs directly
3. **Compiler** (`src/tinycl/compiler.py`) - Generates Python and C code

5.2.1 Parser Implementation

The TinyCL parser uses our PEG library to define a complete grammar:

```python from src.peg import ( PEGParser, Rule, Reference, ParseError, Sequence, Choice, ZeroOrMore, OneOrMore, Optional, Literal, Regex ) from src.peg.syntax_tree import GrammarNode from src.tinycl.ast import *

class TinyCLParser(PEGParser): """Parser for the TinyCL language - Complete Implementation."""
```

```
def __init__(self):
    super().__init__()

    # Define complete grammar for TinyCL
    self.grammar = GrammarNode(
        name="TinyCL",
        rules=[
            # Program structure
            Rule("Program", ZeroOrMore(Reference("Statement"))),

            # Statements - All implemented features
            Rule("Statement", Choice(
                Reference("FunctionDecl"),
                Reference("VariableDecl"),
                Reference("ConstantDecl"),
                Reference("IfStatement"),
                Reference("WhileStatement"),
                Reference("PrintStatement"),
                Reference("ReturnStatement"),
                Reference("AssignmentStatement"),
                Reference("ExpressionStatement"),
                Reference("Block"),
                Reference("Comment")
            )),

            # Function declaration - Fully implemented
            Rule("FunctionDecl", Sequence(
                Literal("func"),
                Reference("Identifier"),
                Literal("("),
                Optional(Reference("Parameters")),
                Literal(")"),
                Reference("Block")
            )),

            # Parameters - Supports multiple parameters
            Rule("Parameters", Sequence(
```

```
                Reference("Identifier"),
                ZeroOrMore(Sequence(
                    Literal(","),
                    Reference("Identifier")
                ))
            )),

            # Variable declaration - Complete implementation
            Rule("VariableDecl", Sequence(
                Literal("var"),
                Reference("Identifier"),
                Literal("="),
                Reference("Expression"),
                Literal(";")
            )),

            # Constant declaration - Complete implementation
            Rule("ConstantDecl", Sequence(
                Literal("const"),
                Reference("Identifier"),
                Literal("="),
                Reference("Expression"),
                Literal(";")
            )),

            # If statement with else support
            Rule("IfStatement", Sequence(
                Literal("if"),
                Literal("("),
                Reference("Expression"),
                Literal(")"),
                Reference("Block"),
                Optional(Sequence(
                    Literal("else"),
                    Reference("Block")
                ))
            )),
```

```
# While statement - Complete implementation
Rule("WhileStatement", Sequence(
    Literal("while"),
    Literal("("),
    Reference("Expression"),
    Literal(")"),
    Reference("Block")
)),

# Print statement - Built-in function
Rule("PrintStatement", Sequence(
    Literal("print"),
    Literal("("),
    Reference("Expression"),
    Literal(")"),
    Literal(";")
)),

# Return statement - Complete implementation
Rule("ReturnStatement", Sequence(
    Literal("return"),
    Optional(Reference("Expression")),
    Literal(";")
)),

# Block - Supports nested statements
Rule("Block", Sequence(
    Literal("{"),
    ZeroOrMore(Reference("Statement")),
    Literal("}")
)),

# Expression hierarchy with proper precedence
Rule("Expression", Reference("LogicalOr")),

# Logical OR - Complete implementation
```

```
        Rule("LogicalOr", Sequence(
            Reference("LogicalAnd"),
            ZeroOrMore(Sequence(
                Literal("||"),
                Reference("LogicalAnd")
            ))
        )),

        # Logical AND - Complete implementation
        Rule("LogicalAnd", Sequence(
            Reference("Equality"),
            ZeroOrMore(Sequence(
                Literal("&&"),
                Reference("Equality")
            ))
        )),

        # Equality operators
        Rule("Equality", Sequence(
            Reference("Comparison"),
            ZeroOrMore(Sequence(
                Choice(
                    Literal("!="),
                    Literal("==")
                ),
                Reference("Comparison")
            ))
        )),

        # Comparison operators
        Rule("Comparison", Sequence(
            Reference("Term"),
            ZeroOrMore(Sequence(
                Choice(
                    Literal("<="),
                    Literal(">="),
                    Literal("<"),
```

```
                Literal(">")
            ),
            Reference("Term")
        ))
    )),

    # Arithmetic: Addition and Subtraction
    Rule("Term", Sequence(
        Reference("Factor"),
        ZeroOrMore(Sequence(
            Choice(
                Literal("+"),
                Literal("-")
            ),
            Reference("Factor")
        ))
    )),

    # Arithmetic: Multiplication and Division
    Rule("Factor", Sequence(
        Reference("Unary"),
        ZeroOrMore(Sequence(
            Choice(
                Literal("*"),
                Literal("/")
            ),
            Reference("Unary")
        ))
    )),

    # Unary operators
    Rule("Unary", Choice(
        Sequence(
            Choice(
                Literal("!"),
                Literal("-")
            ),
```

```
                Reference("Unary")
            ),
            Reference("Postfix")
        )),

        # Postfix: Array access
        Rule("Postfix", Sequence(
            Reference("Primary"),
            ZeroOrMore(Sequence(
                Literal("["),
                Reference("Expression"),
                Literal("]")
            ))
        )),

        # Primary expressions
        Rule("Primary", Choice(
            Sequence(
                Literal("("),
                Reference("Expression"),
                Literal(")")
            ),
            Sequence(
                Reference("Identifier"),
                Literal("("),
                Optional(Reference("Arguments")),
                Literal(")")
            ),
            Sequence(
                Literal("["),
                Optional(Reference("Arguments")),
                Literal("]")
            ),
            Reference("Identifier"),
            Reference("Number"),
            Reference("String"),
            Reference("Character"),
```

```
                Literal("true"),
                Literal("false")
            )),

            # Arguments
            Rule("Arguments", Sequence(
                Reference("Expression"),
                ZeroOrMore(Sequence(
                    Literal(","),
                    Reference("Expression")
                ))
            )),

            # Terminals - All data types
            Rule("Number", Regex("[0-9]+")),
            Rule("String", Regex("\"[^\"]*\"")),
            Rule("Character", Regex("'[^']*'")),
            Rule("Identifier", Regex("[a-zA-Z_][a-zA-Z0-9_]*")),
            Rule("Comment", Regex("#[^\n]*"))
        ]
    )

def skip_whitespace(self, ctx):
    """Skip whitespace and comments."""
    while not ctx.eof():
        if ctx.peek() in " \t\n\r":
            ctx.consume()
            continue
        if ctx.peek() == '#':
            while not ctx.eof() and ctx.peek() != '\n':
                ctx.consume()
            continue
        break

def _parse_expression(self, expr, ctx):
    """Override to handle whitespace between tokens."""
    self.skip_whitespace(ctx)
```

```python
        result = super()._parse_expression(expr, ctx)
        self.skip_whitespace(ctx)
        return result

    def parse(self, text):
        """Parse a TinyCL program and build an AST."""
        result = super().parse(text)
        return self._build_ast(result)

    def _build_ast(self, parse_result):
        """Build an AST from the parse result."""
        if parse_result is None:
            return None
        return ProgramNode(self._build_statements(parse_result))

    # ... (AST building methods implemented in actual parser)
```

```

This parser defines the grammar for TinyCL and implements the `parse` method to parse TinyCL programs. However, it doesn't yet build an AST or interpret the programs.

### 5.2.1 Lexical Elements

Let's enhance our parser to handle lexical elements like whitespace and comments properly:

```python

# Add to TinyCLParser class

def skip_whitespace(self, ctx): """Skip whitespace and comments.""" while not ctx.eof(): # Skip whitespace if ctx.peek().isspace(): ctx.consume() continue

```python
        # Skip comments
        if ctx.peek() == '#':
            while not ctx.eof() and ctx.peek() != '\n':
```

```
            ctx.consume()
        continue

    # No more whitespace or comments to skip
    break
```

```

### 5.2.2 Expressions

Now, let's implement the parsing of expressions:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class

def parse_expression(self, ctx): """Parse an expression.""" self.skip_whitespace(ctx)

```
# Parse a term
left = self.parse_term(ctx)
if left is None:
    return None

# Parse any following +/- operations
while not ctx.eof():
    self.skip_whitespace(ctx)

    # Try to parse an operator
    op_pos = ctx.pos
    if ctx.peek() == '+' or ctx.peek() == '-':
        op = ctx.consume()
```

```
        self.skip_whitespace(ctx)

        # Parse the right term
        right = self.parse_term(ctx)
        if right is None:
            # Backtrack if the right term fails
            ctx.pos = op_pos
            break

        # Create a binary operation node
        left = BinaryOpNode(op, left, right)
    else:
        break

return left
```

def parse_term(self, ctx): """Parse a term.""" self.skip_whitespace(ctx)

```
# Parse a factor
left = self.parse_factor(ctx)
if left is None:
    return None

# Parse any following */÷ operations
while not ctx.eof():
    self.skip_whitespace(ctx)

    # Try to parse an operator
    op_pos = ctx.pos
    if ctx.peek() == '*' or ctx.peek() == '/':
        op = ctx.consume()

        self.skip_whitespace(ctx)

        # Parse the right factor
        right = self.parse_factor(ctx)
```

```
        if right is None:
            # Backtrack if the right factor fails
            ctx.pos = op_pos
            break

        # Create a binary operation node
        left = BinaryOpNode(op, left, right)
    else:
        break

return left
```

def parse_factor(self, ctx): """Parse a factor.""" self.skip_whitespace(ctx)

```
if ctx.eof():
    return None

# Try to parse a number
if ctx.peek().isdigit():
    return self.parse_number(ctx)

# Try to parse a string
if ctx.peek() == '"':
    return self.parse_string(ctx)

# Try to parse an identifier
if ctx.peek().isalpha() or ctx.peek() == '_':
    return self.parse_identifier(ctx)

# Try to parse a parenthesized expression
if ctx.peek() == '(':
    ctx.consume()  # Consume '('

    self.skip_whitespace(ctx)

    # Parse the inner expression
    expr = self.parse_expression(ctx)
```

```
    if expr is None:
        return None

    self.skip_whitespace(ctx)

    # Expect a closing parenthesis
    if ctx.eof() or ctx.peek() != ')':
        return None
    ctx.consume()  # Consume ')'

    return expr

return None
```

def parse_number(self, ctx): """Parse a number.""" start = ctx.pos while not ctx.eof() and ctx.peek().isdigit(): ctx.consume()

```
if start == ctx.pos:
    return None

return NumberNode(int(ctx.text[start:ctx.pos]))
```

def parse_string(self, ctx): """Parse a string.""" if ctx.eof() or ctx.peek() != '"': return None

```
ctx.consume()  # Consume opening quote

start = ctx.pos
while not ctx.eof() and ctx.peek() != '"':
    ctx.consume()

if ctx.eof():
    return None  # Unterminated string

value = ctx.text[start:ctx.pos]
ctx.consume()  # Consume closing quote
```

```
return StringNode(value)
```

def parse_identifier(self, ctx): """Parse an identifier.""" if ctx.eof() or not (ctx.peek().isalpha() or ctx.peek() == '_'): return None

```
start = ctx.pos
ctx.consume()  # Consume first character

while not ctx.eof() and (ctx.peek().isalnum() or ctx.peek() == '_'):
    ctx.consume()

name = ctx.text[start:ctx.pos]
return IdentifierNode(name)
```

```

### 5.2.3 Statements

Now, let's implement the parsing of statements:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class

def parse_statement(self, ctx): """Parse a statement.""" self.skip_whitespace(ctx)

```
if ctx.eof():
    return None

Try to parse a var statement
```

```
if ctx.pos + 3 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+3] == "var"
    return self.parse_var_statement(ctx)

# Try to parse an if statement
if ctx.pos + 2 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+2] == "if":
    return self.parse_if_statement(ctx)

# Try to parse a while statement
if ctx.pos + 5 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+5] == "whil
    return self.parse_while_statement(ctx)

# Try to parse a print statement
if ctx.pos + 5 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+5] == "prin
    return self.parse_print_statement(ctx)

# Try to parse a block
if ctx.peek() == '{':
    return self.parse_block(ctx)

# Try to parse an assignment statement
return self.parse_assignment_statement(ctx)
```

def parse_var_statement(self, ctx): """Parse a var statement.""" start_pos = ctx.pos

```
# Expect "var"
if ctx.pos + 3 > len(ctx.text) or ctx.text[ctx.pos:ctx.pos+3] != "var":
    return None
ctx.pos += 3

self.skip_whitespace(ctx)

# Parse identifier
identifier = self.parse_identifier(ctx)
if identifier is None:
    ctx.pos = start_pos
    return None
```

```
self.skip_whitespace(ctx)

# Expect "="
if ctx.eof() or ctx.peek() != '=':
    ctx.pos = start_pos
    return None
ctx.consume()

self.skip_whitespace(ctx)

# Parse expression
expression = self.parse_expression(ctx)
if expression is None:
    ctx.pos = start_pos
    return None

self.skip_whitespace(ctx)

# Expect ";"
if ctx.eof() or ctx.peek() != ';':
    ctx.pos = start_pos
    return None
ctx.consume()

return VariableDeclNode(identifier, expression)
```

Similar implementations for parse_assignment_statement, parse_if_statement,

parse_while_statement, parse_print_statement, and parse_block

```

#### 5.2.4 Program Structure

Finally, let's implement the parsing of the overall program structure:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class

def parse_program(self, ctx): """Parse a program.""" statements = []

```
while not ctx.eof():
    self.skip_whitespace(ctx)

    if ctx.eof():
        break
```

```
    statement = self.parse_statement(ctx)
    if statement is None:
        break

    statements.append(statement)

return ProgramNode(statements)
```

def parse(self, text: str): """Parse a TinyCL program.""" print(f"Parsing TinyCL program:\n{text}")

```
# Create a ParserContext
ctx = ParserContext(text)

# Parse the program
program = self.parse_program(ctx)

# Skip any trailing whitespace
self.skip_whitespace(ctx)

# Check if we consumed all input
if ctx.eof():
    return program
else:
    raise ParseError(f"Unexpected input at position {ctx.pos}: '{ctx.tex
```

```

## 5.3 Building the Abstract Syntax Tree

Now that we have the parsing functions, we need to define the AST node classes:

```python

# Note: In practice, these AST classes are defined in src/tinycl/ast.py

# and imported with: from src.tinycl.ast import *

class ASTNode: """Base class for AST nodes.""" pass

class ProgramNode(ASTNode): """AST node for a program.""" def **init**(self, statements): self.statements = statements

class StatementNode(ASTNode): """Base class for statement nodes.""" pass

class VariableDeclNode(StatementNode): """AST node for a variable declaration.""" def **init**(self, identifier, expression): self.identifier = identifier self.expression = expression

class AssignStatementNode(StatementNode): """AST node for an assignment statement.""" def **init**(self, identifier, expression): self.identifier = identifier self.expression = expression

class IfStatementNode(StatementNode): """AST node for an if statement.""" def **init**(self, condition, then_statement, else_statement=None): self.condition = condition self.then_statement = then_statement self.else_statement = else_statement

class WhileStatementNode(StatementNode): """AST node for a while statement.""" def **init**(self, condition, body): self.condition = condition self.body = body

class PrintStatementNode(StatementNode): """AST node for a print statement.""" def **init**(self, expression): self.expression = expression

class BlockNode(StatementNode): """AST node for a block.""" def **init**(self, statements): self.statements = statements

class ExpressionNode(ASTNode): """Base class for expression nodes.""" pass

class BinaryOpNode(ExpressionNode): """AST node for a binary operation.""" def **init**(self, op, left, right): self.op = op self.left = left self.right = right class NumberNode(ExpressionNode): """AST node for a number.""" def **init**(self, value): self.value = value

class StringNode(ExpressionNode): """AST node for a string.""" def **init**(self, value): self.value = value

class IdentifierNode(ExpressionNode): """AST node for an identifier.""" def **init**(self, name): self.name = name

class ConditionNode(ASTNode): """AST node for a condition.""" def **init**(self, left, op, right): self.left = left self.op = op self.right = right ```

### 5.3.1 AST Node Classes

These AST node classes represent the different elements of a TinyCL program. Each class corresponds to a specific grammar rule and contains the necessary information to represent that element in the AST.

### 5.3.2 Tree Construction

The parsing functions we implemented earlier construct the AST as they parse the input. Each parsing function returns an AST node representing the parsed element.

## 5.4 Semantic Analysis

After parsing the program and building the AST, we need to perform semantic analysis to check for errors and prepare for interpretation.

### 5.4.1 Symbol Table

The symbol table keeps track of variables and their types:

```python class SymbolTable: """Symbol table for tracking variables.""" def **init**(self): self.symbols = {}

```
def define(self, name, value):
    """Define a variable."""
    self.symbols[name] = value

def lookup(self, name):
```
```

~~~
    """Look up a variable."""
    return self.symbols.get(name)

def update(self, name, value):
    """Update a variable's value."""
    if name not in self.symbols:
        raise NameError(f"Variable '{name}' not defined")
    self.symbols[name] = value
~~~

```

### 5.4.2 Type Checking

We can add type checking to ensure that operations are performed on compatible types:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

def check_types(node, symbol_table): """Check types in the AST.""" if isinstance(node, ProgramNode): for statement in node.statements: check_types(statement, symbol_table)

~~~
elif isinstance(node, VariableDeclNode):
    # Check that the expression has a valid type
    expr_type = get_expression_type(node.expression, symbol_table)
    if expr_type is None:
        raise TypeError(f"Invalid expression in variable declaration")

    # Define the variable
    symbol_table.define(node.identifier.name, None)

elif isinstance(node, AssignStatementNode):
~~~

```
    # Check that the variable is defined
    if symbol_table.lookup(node.identifier.name) is None:
        raise NameError(f"Variable '{node.identifier.name}' not defined'

    # Check that the expression has a valid type
    expr_type = get_expression_type(node.expression, symbol_table)
    if expr_type is None:
        raise TypeError(f"Invalid expression in assignment")

Similar implementations for other node types

return True
```

def get_expression_type(node, symbol_table): """Get the type of an expression.""" if isinstance(node, NumberNode): return "number"

```
elif isinstance(node, StringNode):
    return "string"

elif isinstance(node, IdentifierNode):
    # Check that the variable is defined
    if symbol_table.lookup(node.name) is None:
        raise NameError(f"Variable '{node.name}' not defined")

    # Return the type of the variable
    value = symbol_table.lookup(node.name)
    if isinstance(value, int):
        return "number"
    elif isinstance(value, str):
        return "string"
    else:
        return None

elif isinstance(node, BinaryOpNode):
    # Get the types of the operands
    left_type = get_expression_type(node.left, symbol_table)
    right_type = get_expression_type(node.right, symbol_table)
```

```
    # Check that the operation is valid for the operand types
    if node.op in ['+', '-', '*', '/']:
        if left_type == "number" and right_type == "number":
            return "number"
        elif node.op == '+' and (left_type == "string" or right_type ==
            return "string"  # String concatenation
        else:
            raise TypeError(f"Invalid operand types for operator '{node.
    else:
        raise TypeError(f"Unknown operator: {node.op}")

# Similar implementations for other node types

return None
```

```

## 5.5 Interpreter Implementation

Now, let's implement the interpreter that will execute TinyCL programs:

```python from src.tinycl.ast import *

class TinyCLInterpreter: """Interpreter for TinyCL programs.""" def **init**(self): self.symbol_table = SymbolTable()

```
def interpret(self, program):
    """Interpret a TinyCL program."""
    if not isinstance(program, ProgramNode):
        raise TypeError("Expected a ProgramNode")

    # Perform semantic analysis
    check_types(program, self.symbol_table)

    # Execute the program
    return self.execute_program(program)
```

```
def execute_program(self, program):
    """Execute a program."""
    result = None
    for statement in program.statements:
        result = self.execute_statement(statement)
    return result

def execute_statement(self, statement):
    """Execute a statement."""
    if isinstance(statement, VariableDeclNode):
        return self.execute_var_statement(statement)
    elif isinstance(statement, AssignStatementNode):
        return self.execute_assign_statement(statement)
    elif isinstance(statement, IfStatementNode):
        return self.execute_if_statement(statement)
    elif isinstance(statement, WhileStatementNode):
        return self.execute_while_statement(statement)
    elif isinstance(statement, PrintStatementNode):
        return self.execute_print_statement(statement)
    elif isinstance(statement, BlockNode):
        return self.execute_block(statement)
    else:
        raise TypeError(f"Unknown statement type: {type(statement)}")

def execute_var_statement(self, statement):
    """Execute a variable declaration statement."""
    value = self.evaluate_expression(statement.expression)
    self.symbol_table.define(statement.identifier.name, value)
    return None

def execute_assign_statement(self, statement):
    """Execute an assignment statement."""
    value = self.evaluate_expression(statement.expression)
    self.symbol_table.update(statement.identifier.name, value)
    return None

def execute_if_statement(self, statement):
```

```python
    """Execute an if statement."""
    condition = self.evaluate_condition(statement.condition)
    if condition:
        return self.execute_statement(statement.then_statement)
    elif statement.else_statement is not None:
        return self.execute_statement(statement.else_statement)
    return None

def execute_while_statement(self, statement):
    """Execute a while statement."""
    result = None
    while self.evaluate_condition(statement.condition):
        result = self.execute_statement(statement.body)
    return result

def execute_print_statement(self, statement):
    """Execute a print statement."""
    value = self.evaluate_expression(statement.expression)
    print(value)
    return None

def execute_block(self, block):
    """Execute a block."""
    result = None
    for statement in block.statements:
        result = self.execute_statement(statement)
    return result

def evaluate_expression(self, expression):
    """Evaluate an expression."""
    if isinstance(expression, NumberNode):
        return expression.value
    elif isinstance(expression, StringNode):
        return expression.value
    elif isinstance(expression, IdentifierNode):
        return self.symbol_table.lookup(expression.name)
    elif isinstance(expression, BinaryOpNode):
```

```python
            left = self.evaluate_expression(expression.left)
            right = self.evaluate_expression(expression.right)

            if expression.op == '+':
                return left + right
            elif expression.op == '-':
                return left - right
            elif expression.op == '*':
                return left * right
            elif expression.op == '/':
                return left / right
            else:
                raise ValueError(f"Unknown operator: {expression.op}")
        else:
            raise TypeError(f"Unknown expression type: {type(expression)}")

    def evaluate_condition(self, condition):
        """Evaluate a condition."""
        left = self.evaluate_expression(condition.left)
        right = self.evaluate_expression(condition.right)

        if condition.op == '==':
            return left == right
        elif condition.op == '!=':
            return left != right
        elif condition.op == '<':
            return left < right
        elif condition.op == '>':
            return left > right
        elif condition.op == '<=':
            return left <= right
        elif condition.op == '>=':
            return left >= right
        else:
            raise ValueError(f"Unknown comparison operator: {condition.op}")
```

5.5.1 Runtime Environment

The `SymbolTable` class provides the runtime environment for TinyCL programs. It keeps track of variables and their values.

5.5.2 Expression Evaluation

The `evaluate_expression` method evaluates expressions by recursively evaluating their components and applying the appropriate operations.

5.5.3 Statement Execution

The `execute_statement` method executes statements by dispatching to the appropriate method based on the statement type.

5.6 Example Programs and Testing

Let's create some example TinyCL programs to test our implementation:

```python

# Example 1: Factorial

factorial_program = """

# Calculate factorial

var n = 5; var factorial = 1;

while (n > 0) { factorial = factorial * n; n = n - 1; }

print("Factorial: " + factorial); """

# Example 2: Fibonacci

fibonacci_program = """
```

Calculate Fibonacci numbers

var n = 10; var a = 0; var b = 1; var i = 0;

print("Fibonacci sequence:"); print(a); print(b);

while (i < n - 2) { var c = a + b; print(c); a = b; b = c; i = i + 1; } """

Test the TinyCL interpreter

if **name** == "**main**": parser = TinyCLParser() interpreter = TinyCLInterpreter()

```
print("Testing factorial program:")
try:
    ast = parser.parse(factorial_program)
    interpreter.interpret(ast)
except Exception as e:
    print(f"Error: {e}")

print("\nTesting Fibonacci program:")
try:
    ast = parser.parse(fibonacci_program)
    interpreter.interpret(ast)
except Exception as e:
    print(f"Error: {e}")
```

```

## Summary

In this chapter, we've built a complete tiny programming language called TinyCL using our PEG parser library. We've:

1. Designed the language features and grammar
2. Implemented the parser to build an AST
3. Added semantic analysis for type checking
4. Created an interpreter to execute TinyCL programs
```

5. Tested the implementation with example programs

TinyCL demonstrates the power and flexibility of the TinyPEG library for building parsers and interpreters. While it's a simple language, it includes many of the fundamental concepts found in larger programming languages.

In the next chapter, we'll explore advanced topics and extensions to both the TinyPEG library and the TinyCL language.