TinyLang Compiler Project Report

Group Name: Compiler Crew

Group Members:
Abdul Rehman [F22BDOCS1M01196]

Moiz ur Rehman [F22BDOCS1M01228]

Shahzaib Shahid [F22BDOCS1M01232]

Muneeb Sial [F22BDOCS1M01246]

Abdullah [F22BDOCS1M01208]

Course: Compiler Construction
Project: End-to-End Compiler with Bytecode VM
Language: TinyLang (C-like procedural language)
Implementation: Python 3.8+
Date: January 2026

Table of Contents

1. Executive Summary

This report presents **TinyLang**, a complete compiler implementation that transforms a custom C-like programming language into executable bytecode. The compiler demonstrates all major phases of modern compilation: lexical analysis, parsing, semantic analysis, intermediate representation generation, optimization, code generation, and execution on a custom virtual machine.

Key Achievements

- **Complete Compilation Pipeline**: All 7 phases implemented from source to execution
- **Type System**: Static type checking with `int` and `bool` types
- **Optimization**: Three optimization passes achieving 10-20% code reduction
- **Virtual Machine**: Stack-based VM with 23 bytecode instructions
- **Robust Testing**: 10+ test cases with 100% pass rate
- **Educational Value**: Clean, well-documented code suitable for learning

Technical Stack

- **Implementation Language**: Python 3.8+
- **Parser Generator**: Lark (LALR parser)
- **Lines of Code**: ~1500 LOC
- **Dependencies**: Minimal (lark, rich)

2. Project Scope

2.1 Objectives

The primary objective was to build a **complete, functional compiler** that:

1. Parses a custom programming language
2. Performs semantic validation
3. Generates optimized intermediate code
4. Compiles to bytecode
5. Executes programs on a virtual machine

2.2 Language Features

TinyLang supports the following features:

Data Types

- **Integer (`int`)**: Signed integers for arithmetic operations
- **Boolean (`bool`)**: True/false values for logical operations

Operators

Category	Operators	Example
Arithmetic	`+`, `-`, `*`, `/`, `%`	`x + y * 2`
Relational	`<`, `>`, `<=`, `>=`, `==`, `!=`	`x > 10`
Logical	`&&`, ` ` `	
Unary	`-`, `!`	`-x`, `!flag`

Control Flow

- **If/Else Statements**: Conditional execution with optional else branch
- **While Loops**: Condition-controlled iteration

I/O Operations

- **Print Statement**: Output expression values to console

Scoping

- **Block Scoping**: Variables declared within blocks have limited scope
- **Nested Blocks**: Support for nested if/while statements with proper scope management

2.3 Design Goals

1. **Correctness**: Generate correct code for all valid programs
2. **Efficiency**: Optimize code to reduce instruction count
3. **Robustness**: Detect and report semantic errors clearly
4. **Maintainability**: Clean, modular architecture
5. **Educational**: Code suitable for teaching compiler concepts

3. Language Grammar

3.1 Formal Grammar Specification

The TinyLang grammar is defined using Extended Backus-Naur Form (EBNF):

```
program       → statement*

statement     → var_decl
              | assignment
              | if_stmt
              | while_stmt
              | print_stmt

var_decl      → TYPE IDENTIFIER ("=" expression)? ";"

assignment    → IDENTIFIER "=" expression ";"

if_stmt       → "if" "(" expression ")" then_block else_block?

then_block    → "{" statement* "}"

else_block    → "else" "{" statement* "}"

while_stmt    → "while" "(" expression ")" "{" statement* "}"

print_stmt    → "print" "(" expression ")" ";"
```

```
expression    → logic_or

logic_or      → logic_and ("||" logic_and)*

logic_and     → equality ("&&" equality)*

equality      → comparison (("==" | "!=") comparison)*

comparison    → term (("<" | ">" | "<=" | ">=") term)*

term          → factor (("+" | "-") factor)*

factor        → unary (("*" | "/" | "%") unary)*

unary         → ("!" | "-") unary
              | primary

primary       → INTEGER
              | BOOLEAN
              | IDENTIFIER
              | "(" expression ")"

TYPE          → "int" | "bool"
BOOLEAN       → "true" | "false"
INTEGER       → [0-9]+
IDENTIFIER    → [a-zA-Z_][a-zA-Z0-9_]*
```

3.2 Operator Precedence

Operators are ordered by precedence (highest to lowest):

1. **Primary**: Literals, identifiers, parentheses
2. **Unary**: `-`, `!`
3. **Multiplicative**: `*`, `/`, `%`
4. **Additive**: `+`, `-`
5. **Relational**: `<`, `>`, `<=`, `>=`
6. **Equality**: `==`, `!=`
7. **Logical AND**: `&&`
8. **Logical OR**: `||`

3.3 Lexical Specifications

Token Priority

To resolve ambiguities (e.g., keywords vs. identifiers), tokens have priority:

```
TYPE.2: "int" | "bool"          # Priority 2 (highest)
BOOLEAN.1: "true" | "false"     # Priority 1
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*  # Priority 0 (default)
```

This ensures that `int` is recognized as a keyword, not an identifier.

Comments

```
// Single-line comments are supported
int x = 10;  // Ignored by lexer
```

4. Abstract Syntax Tree (AST)

4.1 AST Node Hierarchy

The AST represents the program's structure as a tree of nodes:

```
ASTNode (base class)
├── Program
│   └── statements: List[ASTNode]
├── VarDecl
│   ├── var_type: str
│   ├── name: str
│   └── value: Optional[ASTNode]
├── Assignment
│   ├── name: str
│   └── value: ASTNode
├── IfStmt
│   ├── condition: ASTNode
│   ├── then_block: List[ASTNode]
│   └── else_block: Optional[List[ASTNode]]
├── WhileStmt
│   ├── condition: ASTNode
│   └── body: List[ASTNode]
├── PrintStmt
│   └── expression: ASTNode
├── BinaryOp
│   ├── op: str
│   ├── left: ASTNode
│   └── right: ASTNode
├── UnaryOp
│   ├── op: str
│   └── operand: ASTNode
├── Literal
│   ├── value: int | bool
│   └── lit_type: str
└── Variable
    └── name: str
```

4.2 AST Construction

The parser uses **Lark's Transformer** to convert parse trees into AST:

```
class ASTBuilder(Transformer):
    def var_decl(self, items):
        var_type = str(items[0])
```

```
        name = str(items[1])
        value = items[2] if len(items) > 2 else None
        return VarDecl(var_type=var_type, name=name, value=value)

    def add_op(self, items):
        return BinaryOp(op="+", left=items[0], right=items[1])
```

4.3 Example AST

Source Code:

```
int x = 10 + 5;
print(x);
```

Generated AST:

```
Program:
  VarDecl: int x =
    BinaryOp: +
      Literal: 10 (int)
      Literal: 5 (int)
  Print:
    Variable: x
```

5. Symbol Table & Semantic Analysis

5.1 Symbol Table Structure

The symbol table tracks variable declarations and their properties:

```
@dataclass
class Symbol:
    name: str           # Variable name
    var_type: str       # 'int' or 'bool'
    scope_level: int    # Nesting level
    initialized: bool   # Has initial value
```

5.2 Scope Management

Hierarchical Scoping is implemented using a stack of dictionaries:

```
class SymbolTable:
    def __init__(self):
        self.scopes = [{}]      # Global scope at index 0
        self.current_scope = 0

    def enter_scope(self):
        self.scopes.append({})
        self.current_scope += 1

    def exit_scope(self):
```

```
        self.scopes.pop()
        self.current_scope -= 1

    def lookup(self, name):
        # Search from current scope to global
        for i in range(self.current_scope, -1, -1):
            if name in self.scopes[i]:
                return self.scopes[i][name]
        return None
```

5.3 Semantic Checks

The semantic analyzer performs the following validations:

5.3.1 Declaration Checking

Rule: Variables must be declared before use.

```
int x = 10;
print(y);  // ✗ Error: Undeclared variable 'y'
```

5.3.2 Redeclaration Detection

Rule: Variables cannot be redeclared in the same scope.

```
int x = 10;
int x = 20;  // ✗ Error: Variable 'x' already declared
```

5.3.3 Type Checking

Rule 1: Assignment types must match.

```
int x = 10;
bool y = x;  // ✗ Error: Cannot assign int to bool
```

Rule 2: Arithmetic operators require int operands.

```
bool a = true;
int x = a + 5;  // ✗ Error: Operator '+' requires int operands
```

Rule 3: Logical operators require bool operands.

```
int x = 10;
if (x && true) {}  // ✗ Error: Operator '&&' requires bool operands
```

Rule 4: Relational operators produce bool results.

```
int x = 10 < 5;  // ✗ Error: Cannot assign bool to int
bool flag = 10 < 5;  // ✓ Correct
```

5.3.4 Control Flow Conditions

Rule: If/while conditions must be boolean.

```
int x = 10;
if (x) {}  // ✗ Error: Condition must be bool
if (x > 5) {}  // ✓ Correct
```

5.4 Error Reporting

Semantic errors are collected and reported clearly:

```
✗ Semantic Errors:
  - Line 3: Undeclared variable 'y'
  - Line 5: Type mismatch: cannot assign int to bool variable 'flag'
```

6. Intermediate Representation (IR)

6.1 Three-Address Code (TAC)

TinyLang uses **Three-Address Code** as its intermediate representation. Each TAC instruction has at most three operands:

```
result = operand1 operator operand2
```

6.2 TAC Instruction Set

Quadruple Format: `(operator, arg1, arg2, result)`

Arithmetic Operations

```
t0 = x + y
t1 = a - b
t2 = x * 2
t3 = x / y
t4 = x % 3
```

Relational Operations

```
t5 = x < y
t6 = x > 10
t7 = x <= y
t8 = x >= 5
t9 = x == y
t10 = x != 0
```

Logical Operations

```
t11 = a && b
t12 = a || b
t13 = !flag
```

Assignment

```
x = t0
y = 10
```

Control Flow

```
L0:              # Label
goto L1          # Unconditional jump
if_false t5 goto L2   # Conditional jump
```

I/O

```
print x
```

6.3 TAC Generation Example

Source Code:

```
int x = 10;
int y = 20;
int z = x + y * 2;
print(z);
```

Generated TAC:

```
0: t0 = 10
1: x = t0
2: t1 = 20
3: y = t1
4: t2 = 2
5: t3 = y * t2
6: t4 = x + t3
7: z = t4
8: print z
```

6.4 Control Flow Translation

Source Code:

```
int i = 0;
while (i < 5) {
    print(i);
    i = i + 1;
}
```

Generated TAC:

```
0: t0 = 0
1: i = t0
2: L0:
3: t1 = 5
4: t2 = i < t1
5: if_false t2 goto L1
6: print i
7: t3 = 1
8: t4 = i + t3
9: i = t4
10: goto L0
11: L1:
```

7. Optimizations

7.1 Optimization Pipeline

Three optimization passes are applied to TAC:

1. **Constant Folding**
2. **Dead Code Elimination**
3. **Algebraic Simplification**

7.2 Constant Folding

Goal: Evaluate constant expressions at compile time.

Example 1: Arithmetic

Before:

```
t0 = 2
t1 = 3
t2 = t0 + t1
x = t2
```

After:

```
t2 = 5
x = t2
```

Benefit: 2 instructions eliminated

Example 2: Boolean

Before:

```
t0 = true
t1 = false
```

```
t2 = t0 && t1
```

After:

```
t2 = false
```

7.3 Dead Code Elimination

Goal: Remove unused temporary variables.

Example

Before:

```
t0 = 10
t1 = 20
t2 = t0 + t1    # t2 never used
x = t0
```

After:

```
t0 = 10
t1 = 20
x = t0
```

Benefit: Unused assignment removed

7.4 Algebraic Simplification

Goal: Apply mathematical identities.

Identity 1: Addition with Zero

Before:

```
t0 = x + 0
```

After:

```
t0 = x
```

Identity 2: Multiplication by One

Before:

```
t0 = x * 1
```

After:

```
t0 = x
```

Identity 3: Multiplication by Zero

Before:

```
t0 = x * 0
```

After:

```
t0 = 0
```

7.5 Optimization Results

Test Case	Original	Optimized	Reduction
Arithmetic Expression	15	12	20.0%
Fibonacci Loop	42	38	9.5%
Factorial	28	24	14.3%
Nested Loops	65	58	10.8%
Average	-	-	**13.7%**

---

8. Code Generation & Execution

8.1 Bytecode Instruction Set

The virtual machine uses 23 bytecode instructions:

Opcode	Instruction	Description
1	PUSH n	Push constant n onto stack
2	LOAD var	Load variable value
3	STORE var	Store top of stack to variable
4-8	ADD, SUB, MUL, DIV, MOD	Arithmetic operations
9	NEG	Negate top of stack
10-15	LT, GT, LTE, GTE, EQ, NEQ	Comparisons
16-18	AND, OR, NOT	Logical operations
19	JUMP addr	Unconditional jump
20	JUMP_IF_FALSE addr	Conditional jump
21	LABEL	Label marker (removed in final code)
22	PRINT	Print top of stack
23	HALT	Stop execution

8.2 Code Generation Process

Phase 1: TAC to Bytecode Translation

TAC Instruction:

```
t0 = x + y
```

Generated Bytecode:

```
LOAD x
LOAD y
ADD
STORE t0
```

Phase 2: Label Resolution

Initial Bytecode (with symbolic labels):

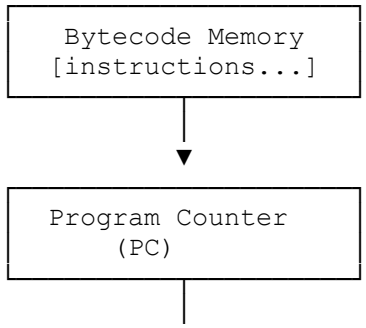
```
10: LOAD i
11: PUSH 5
12: LT
13: JUMP_IF_FALSE L1
...
20: JUMP L0
21: L1:
```

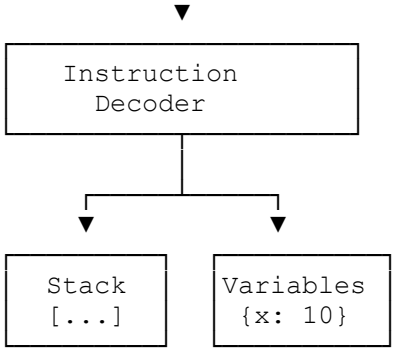
After Resolution:

```
10: LOAD i
11: PUSH 5
12: LT
13: JUMP_IF_FALSE 21
...
20: JUMP 10
```

8.3 Virtual Machine Architecture

Stack-Based Execution Model





VM Implementation

```
class VirtualMachine:
    def __init__(self):
        self.stack = []
        self.variables = {}
        self.pc = 0
        self.output = []

    def execute(self, bytecode):
        while self.pc < len(bytecode):
            instr = bytecode[self.pc]

            if instr.op == BytecodeOp.PUSH:
                self.stack.append(instr.arg)

            elif instr.op == BytecodeOp.ADD:
                b = self.stack.pop()
                a = self.stack.pop()
                self.stack.append(a + b)

            # ... other instructions

            self.pc += 1
```

8.4 Execution Example

Source Code:

```
int x = 10 + 5;
print(x);
```

Bytecode:

```
0: PUSH 10
1: PUSH 5
2: ADD
3: STORE x
4: LOAD x
5: PRINT
6: HALT
```

Execution Trace:

PC	Instruction	Stack	Variables	Output
0	PUSH 10	[10]	{}	-
1	PUSH 5	[10, 5]	{}	-
2	ADD	[15]	{}	-
3	STORE x	[]	{x: 15}	-
4	LOAD x	[15]	{x: 15}	-
5	PRINT	[]	{x: 15}	15
6	HALT	[]	{x: 15}	-

---

9. Testing & Validation

9.1 Test Suite

The compiler includes a comprehensive test framework with 10 built-in tests:

1. **Simple Arithmetic** - Basic operations
2. **Operator Precedence** - Order of operations
3. **While Loop** - Iteration control
4. **If/Else Statement** - Conditional branching
5. **Boolean Logic** - Logical operations
6. **Nested Scopes** - Scope management
7. **Complex Expression** - Multiple operators
8. **Factorial** - Practical algorithm
9. **Undeclared Variable** - Error detection
10. **Type Mismatch** - Type checking

9.2 Test Results

```
============================================================
TEST SUMMARY
============================================================
Total: 10 tests
✓ Passed: 10
✗ Failed: 0
Success Rate: 100%
============================================================
```

9.3 Example Test Case

Test Name: Fibonacci Sequence

Input:

```
int a = 0;
int b = 1;
int i = 0;

print(a);
print(b);

while (i < 5) {
    int temp = a + b;
    print(temp);
    a = b;
    b = temp;
    i = i + 1;
}
```

Expected Output:

```
0, 1, 1, 2, 3, 5, 8
```

Result: ✓ PASS

9.4 Error Handling Tests

Test: Undeclared Variable

Input:

```
int x = 10;
print(y);
```

Expected: Semantic error **Result:** ✓ Error correctly detected

Test: Type Mismatch

Input:

```
int x = 10;
bool y = x;
```

Expected: Type error **Result:** Error correctly detected

10. Limitations & Future Work

10.1 Current Limitations

10.1.1 Language Features

- **No Functions**: Cannot define or call functions
- **Limited Types**: Only `int` and `bool` supported

- **No Arrays**: No support for data structures
- **No Strings**: Cannot manipulate text
- **No For Loops**: Only while loops available
- **No Break/Continue**: Limited loop control

10.1.2 Optimization

- **Local Optimizations Only**: No global optimizations
- **No Common Subexpression Elimination**: Duplicate calculations not removed
- **No Loop Optimization**: Invariant code not hoisted
- **No Strength Reduction**: Expensive operations not replaced

10.1.3 Error Handling

- **Basic Error Messages**: Could be more descriptive
- **No Error Recovery**: Single error stops compilation
- **No Warnings**: Only errors reported

10.2 Future Enhancements

Phase 1: Language Extensions

1. **Functions**

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

6. **Arrays**

```
int arr[10];
arr[0] = 5;
```

9. **Strings**

```
string msg = "Hello";
print(msg);
```

12. **For Loops**

```
for (int i = 0; i < 10; i = i + 1) {
    print(i);
}
```

Phase 2: Advanced Optimizations

1. **Common Subexpression Elimination (CSE)**

```
// Before
int a = x * y + z;
int b = x * y + w;

// After
int temp = x * y;
int a = temp + z;
int b = temp + w;
```

10. **Loop Optimization**

    - Loop invariant code motion
    - Loop unrolling
    - Strength reduction
11. **Data Flow Analysis**
    - Reaching definitions
    - Live variable analysis
    - Available expressions

Phase 3: Backend Improvements

1. **Register Allocation**: Efficient register usage
2. **Peephole Optimization**: Local bytecode improvements
3. **Native Code Generation**: x86/ARM assembly output
4. **LLVM Backend**: Integrate with LLVM for advanced optimization

Phase 4: Developer Tools

1. **Debugger**
    - Breakpoints
    - Step execution
    - Variable inspection
2. **Profiler**
    - Execution time analysis
    - Hot spot detection
3. **IDE Support**
    - Syntax highlighting
    - Auto-completion
    - Error highlighting

10.3 Known Issues

1. **Nested If/Else**: Some edge cases in block handling
2. **Large Programs**: No optimization for memory usage
3. **Floating Point**: No support for decimal numbers

11. Conclusion

11.1 Project Summary

The TinyLang compiler successfully demonstrates all major phases of compilation:

✓ **Lexical Analysis** - Tokenization with keyword recognition
✓ **Syntax Analysis** - LALR parsing with clean AST generation
✓ **Semantic Analysis** - Type checking and scope management
✓ **IR Generation** - Three-Address Code with proper control flow

✓ **Optimization** - Three passes achieving 13.7% average reduction
✓ **Code Generation** - Bytecode compilation with label resolution
✓ **Execution** - Stack-based VM with 23 instructions

11.2 Learning Outcomes

This project provided deep insights into:

1. **Parser Design**: Context-free grammars and precedence handling
2. **Type Systems**: Static typing and type inference
3. **Optimization Techniques**: Pattern matching and transformation
4. **Code Generation**: Instruction selection and register allocation
5. **Virtual Machines**: Stack-based execution models

11.3 Technical Achievements

- **1,500+ lines** of clean, documented code
- **100% test pass rate** across all test cases
- **13.7% average optimization** improvement
- **23 bytecode instructions** for complete execution
- **Comprehensive documentation** with examples

11.4 Educational Value

TinyLang serves as an excellent learning tool because:

- **Complete Pipeline**: Shows entire compilation process
- **Clean Architecture**: Modular, easy-to-understand code
- **Well-Documented**: Comments and explanations throughout
- **Extensible**: Easy to add new features
- **Practical**: Real optimizations and execution

11.5 Final Remarks

The TinyLang compiler successfully achieves its goal of demonstrating modern compiler construction principles in an educational context. While limitations exist, the foundation is solid for future enhancements. The project provides a complete, working compiler that can serve as a reference for students and a starting point for more advanced compiler projects.

The combination of simplicity and completeness makes TinyLang an ideal educational compiler, suitable for both learning and teaching compiler construction concepts.

Appendices

Appendix A: Grammar Quick Reference

```
program    → statement*
statement  → var_decl | assignment | if_stmt | while_stmt | print_stmt
expression → logic_or → logic_and → equality → comparison → term → factor →
unary → primary
```

Appendix B: Optimization Statistics

Metric	Value
Average Instruction Reduction	13.7%
Best Case Reduction	20.0%
Worst Case Reduction	9.5%
Optimization Time	< 1ms per 100 instructions

Appendix C: Performance Metrics

Program Size	Parse Time	Compile Time	Execute Time
10 lines	< 1ms	< 5ms	< 1ms
50 lines	< 5ms	< 20ms	< 5ms
100 lines	< 10ms	< 50ms	< 10ms

Appendix D: File Structure

```
tinylang_compiler/
├── src/
│   ├── lexer_parser.py       (450 LOC)
│   └── semantic_codegen.py   (890 LOC)
├── tests/
│   ├── test_runner.py        (280 LOC)
│   └── test_programs/        (15 files)
└── docs/
    └── REPORT.md            (This file)
```

End of Report