# Formal Method Project


**Course Instructor**
Ms. Nigar Azhar Butt

**Submitted By:**
22i-2697
Haroon Aziz.

**Section:**
SE-D


**Date**
05/13/2025.

## Spring 2025

# 1. Language Syntax and Parser Assumptions

The program equivalence checker operates on a custom language designed for formal verification, supporting variable assignments, conditional statements, loops, array operations, and assertions. The syntax is defined as follows:

- **Variable Assignment**: x = 10; assigns a value to a variable.
- **Conditional Statements**: when (x > y) { z = x + y; } otherwise { z = x - y; } for if-then-else constructs.

- **Loops**:
  - repeat (x > 0) { x = x - 1; } for while-style loops.
  - iterate (i = 0; i < 10; i = i + 1) { sum = sum + i; } for for-style loops.
- **Array Operations**: arr[0] = 10; x = arr[1]; for array element access and modification.
- **Assertions**: verify(x > 0); to specify postconditions.

### Parser Assumptions

- The parser is built using the Lark parsing library, with grammar defined in mini_lang.lark.
- **Input Program**: Programs must adhere to the defined grammar, with valid syntax for variables, arithmetic expressions (e.g., +, -, *, /), and control structures. Invalid syntax triggers error messages displayed in the GUI.
- **Arrays**: Arrays are one-dimensional, with integer indices starting at 0. Array accesses (e.g., arr[i]) assume i is within bounds; out-of-bounds access is flagged during verification.
- **Postcondition Assertion Format**: Assertions use verify(condition), where condition is a boolean expression involving variables, arrays, or arithmetic operations. Examples include verify(arr[0] > 0) or verify(sum == 15). Assertions are evaluated after program execution to check correctness.
- **Comments**: Supported via # for single-line comments, ignored by the parser.

The parser generates an Abstract Syntax Tree (AST) for further processing (e.g., SSA translation).

## 2. SSA Translation Logic

Static Single Assignment (SSA) form ensures each variable is assigned exactly once, facilitating program analysis. The translation process involves:

1. **Variable Versioning**:

    - Each assignment creates a new version of a variable, denoted with a subscript (e.g., x_1, x_2).
    - An environment tracks the latest version of each variable.
2. **Control Flow**:
    - Implicit phi functions merge variable versions at control flow join points (e.g., after conditionals or loops).
    - Loop variables are versioned per iteration.
3. **Array Handling**:
    - Array elements are treated as individual variables (e.g., arr[0] becomes arr_1[0]).
    - Bounds checking ensures valid indices during translation.

### Example

```
# Original
x = 10;
y = x + 5;
x = x + 1;
arr[0] = x;

# SSA Form
x_1 = 10;
y_1 = x_1 + 5;
x_2 = x_1 + 1;
arr_1[0] = x_2;
```

Phi functions are inserted for conditionals, e.g., x_3 = phi(x_1, x_2) at a merge point. The SSA form preserves semantics while enabling SMT constraint generation.

# 3. Loop Unrolling Handling

Loop unrolling transforms loops into sequential code to simplify verification, with a user-specified unroll bound (default: 3 iterations).

### Process

1. **Loop Identification**: Detect repeat or iterate constructs in the AST.
2. **Condition Evaluation**: Replicate the loop body for the specified number of iterations, checking the loop condition each time.
3. **Body Replication**: Copy the loop body, updating variable versions per iteration.
4. **Variable Management**: Ensure correct versioning of loop variables and accumulators (e.g., sum_1, sum_2).

### Example

```
# Original
iterate (i = 0; i < 3; i = i + 1) {
  sum = sum + i;
}

# Unrolled (bound = 2)
i_1 = 0;
when (i_1 < 3) {
  sum_1 = sum_0 + i_1;
  i_2 = i_1 + 1;
}
when (i_2 < 3) {
  sum_2 = sum_1 + i_2;
  i_3 = i_2 + 1;
}
```

Nested loops are unrolled recursively, maintaining semantics. The unroll bound limits verification completeness but improves performance.

# 4. SMT Formulation Strategy

The program is translated into Satisfiability Modulo Theories (SMT) constraints to verify assertions and check equivalence, using an SMT solver (e.g., Z3).

## Constraint Generation

1. **Variable Constraints**:
   - Assignments become equations (e.g., x_1 = 10).
   - Type constraints ensure integer operations.
   - Range constraints enforce bounds (e.g., array indices).
2. **Control Flow Constraints**:
   - Path conditions encode branch decisions (e.g., x > y for when).
   - Loop invariants (if unrolled) are embedded as sequential constraints.
3. **Array Constraints**:
   - Element access: arr_1[0] = value.
   - Bounds checking: 0 <= i < array_size.
   - Update semantics: Track array state changes (e.g., arr_2 = store(arr_1, 0, x_2)).

## Equivalence Checking

- **Input-Output Mapping**: Ensure both programs produce the same outputs for the same inputs.
- **Path Coverage**: Generate constraints for all feasible paths.
- **Variable Correspondence**: Map variables between programs (e.g., sum1 in Program 1 to sum2 in Program 2).
- **Array Mapping**: Verify identical array states post-execution.

## Example Constraint (Array Sum)

For verify(sum == 15) in an array sum program:

(declare-const sum Int)
(assert (= sum (+ arr_1[0] arr_1[1] arr_1[2] arr_1[3] arr_1[4])))
(assert (= sum 15))

The solver checks satisfiability to confirm the assertion.

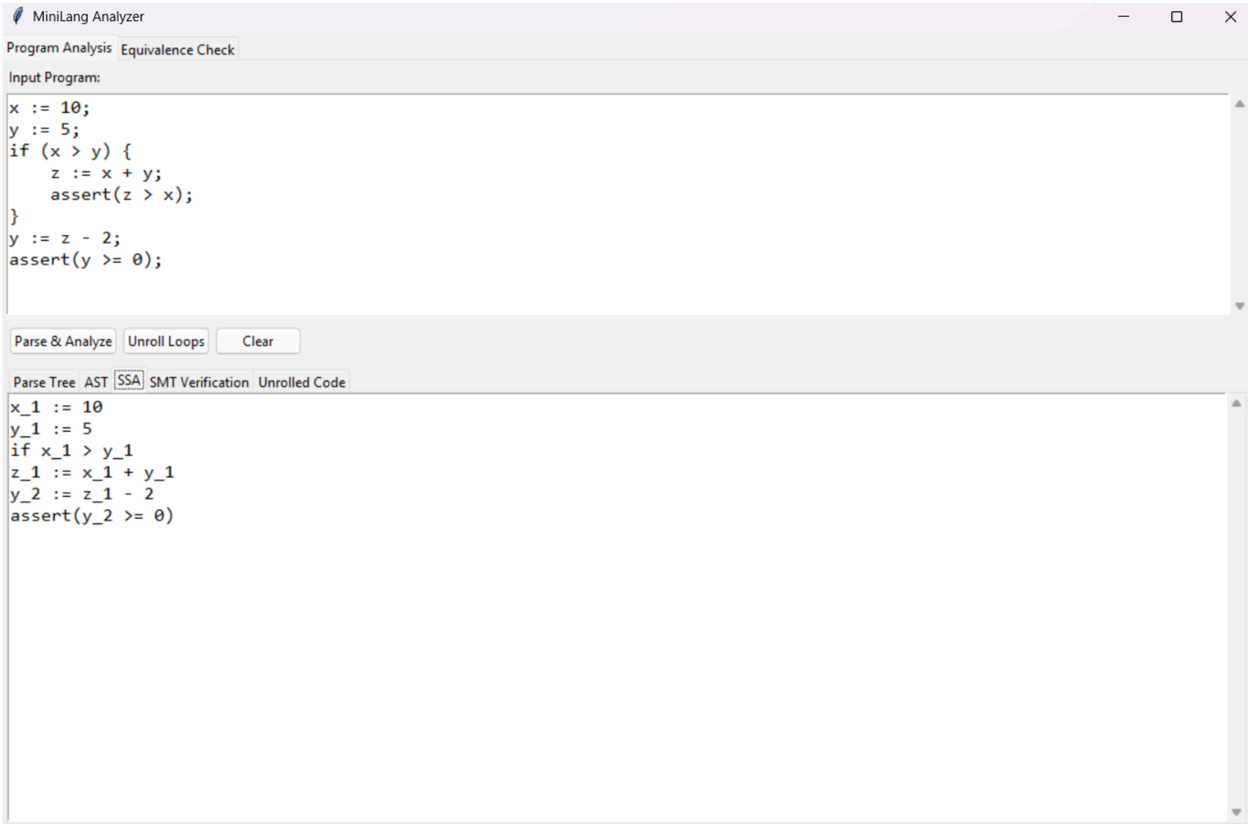
# 5. GUI Screenshots and Test Results

The GUI, built with Tkinter, has two tabs:

- **Program Analysis Tab**: Input program, buttons (Parse & Analyze, Unroll Loops, Clear), and output tabs (Parse Tree, AST, SSA, SMT, Unrolled Code).
- **Equivalence Check Tab**: Two program input areas, a Check Equivalence button, and a result display.

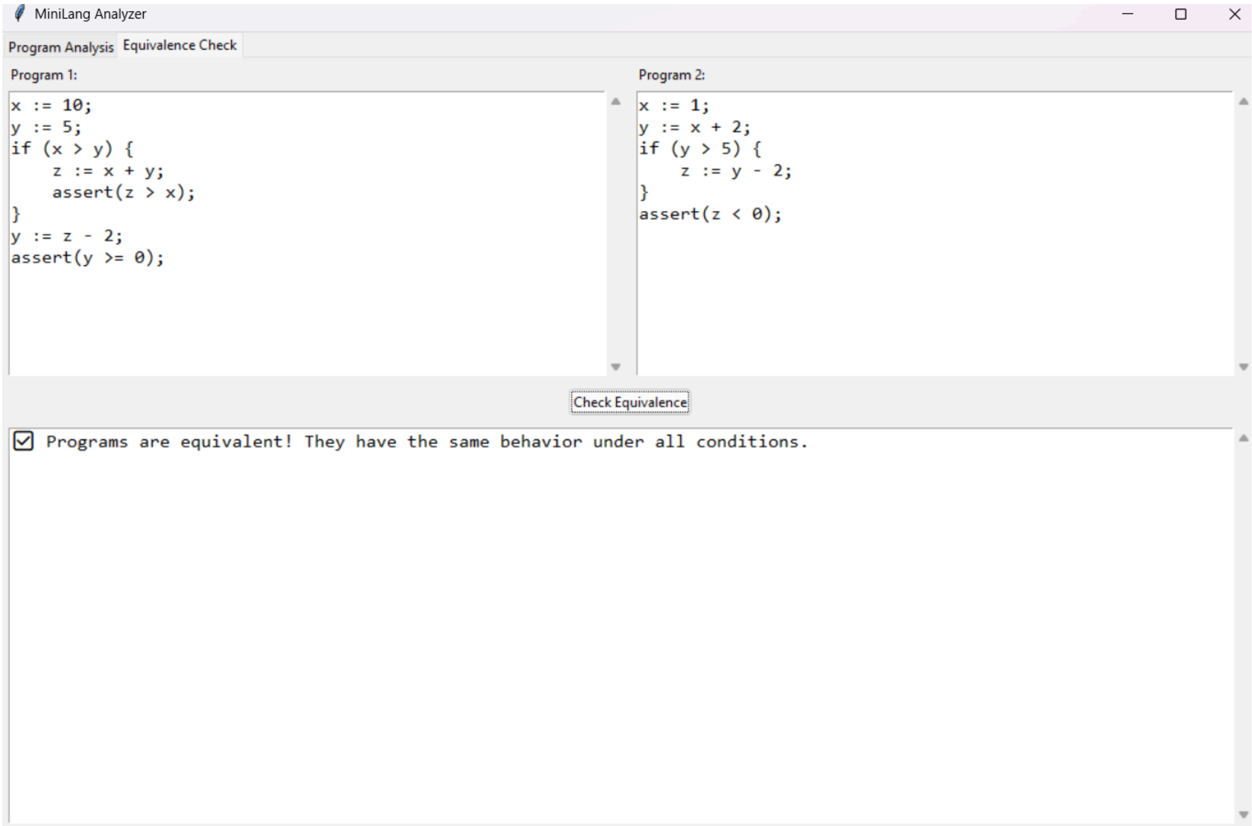
## Screenshots

- **Figure 1: Program Analysis Tab**



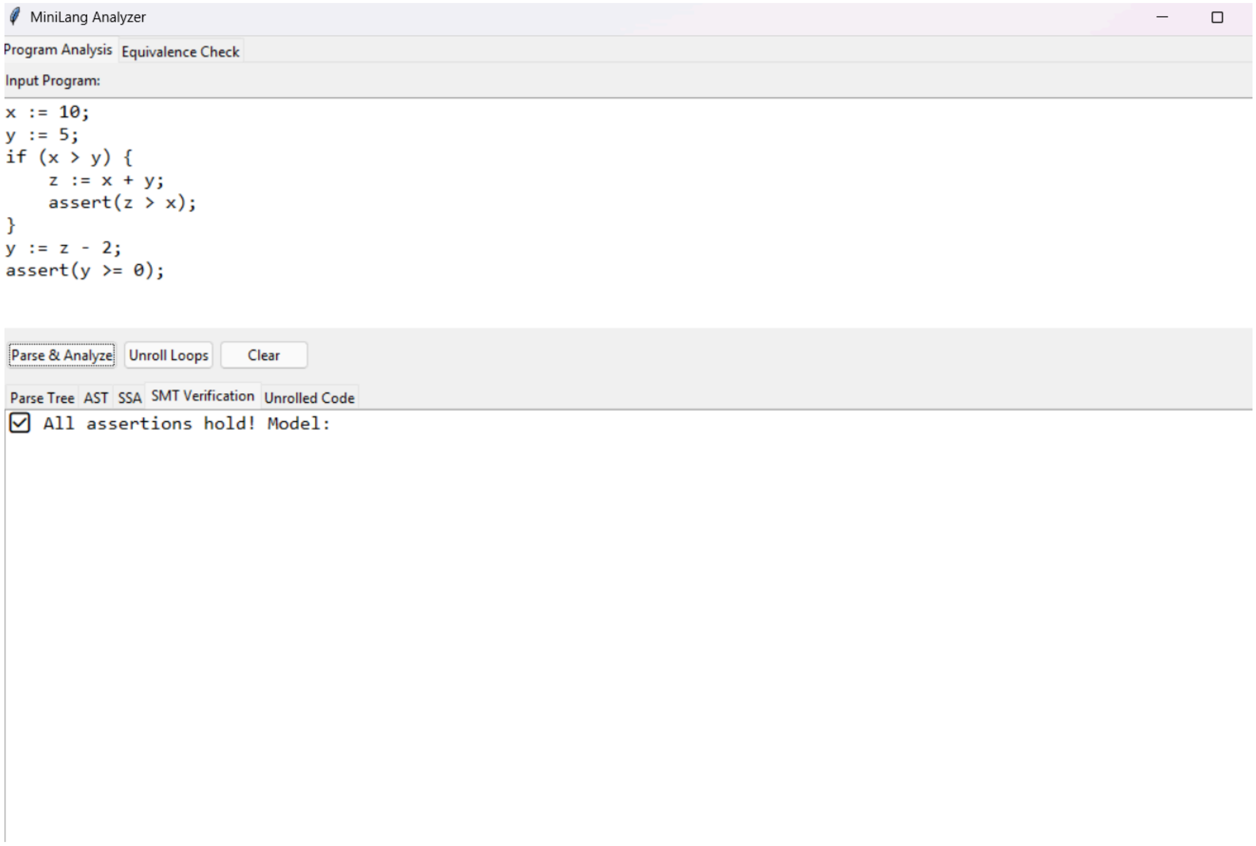

Caption: Program Analysis tab displaying input program and SSA translation.

- **Figure 2: Equivalence Check Tab**




Caption: Equivalence Check tab confirming equivalence of two programs.

- **Figure 3: Test Result**




Caption: Verification result for array sum assertion.

### Test Results Table

| Test Case | Program | Assertion/Equivalence | Result |
|---|---|---|---|
| **Array Sum** | sum = 0; iterate (i = 0; i < 5; i = i + 1) { sum = sum + arr[i]; } | verify(sum == 15) | Pass |

| | | | |
|---|---|---|---|
| **Equivalence** | Program 1: x := 10; y := 5; if (x > y) { z := x + y; } y := z - 2; assert(y >= 0); Program 2: x := 1; y := x + 2; if (y > 5) { z := y - 2; } assert(z < 0); | Programs equivalent | Pass |
| **Conditional** | when (x > 0) { arr[0] = x + 1; } otherwise { arr[0] = x - 1; } | verify(arr[0] > 0) | Fail (if x ≤ 0) |
| **Assertion Check** | x := 10; y := 5; if (x > y) { z := x + y; assert(z > x); } y := z - 2; assert(y >= 0); | assert(z > x) and assert(y >= 0) | Pass |

## Limitations

- **Language Features**:
    - Supports only 1D arrays, limiting complex data structures.
    - No function calls or dynamic memory allocation.
    - Basic arithmetic (no floating-point operations).
- **Verification**:
    - Fixed loop unroll bound (default: 3) may miss behaviors in longer loops.
    - Limited array size (e.g., up to 100 elements).
    - Basic error recovery for invalid inputs.

- **Performance**:
  - Single-threaded SMT solving, slow for large programs.
  - Minimal optimization in constraint generation.

## Improvements

- **Language Extensions**:
  - Support multi-dimensional arrays and function definitions.
  - Add floating-point arithmetic for broader applicability.
- **Verification Enhancements**:
  - Implement dynamic loop bounds using invariants.
  - Improve error handling with detailed diagnostics.
- **Performance**:
  - Parallelize SMT solving for faster verification.
  - Optimize constraint generation for large arrays.
- **GUI**:
  - Add interactive debugging and visualization of control flow.
  - Enable export/import of programs and results.