PROJECT MEMBER NAME

Muskan & Suhail

SUPERVISOR

Ma’am Faryal Shamsi

Distribution in Project
Coding Part is done by Suhail Ahmed
&
Documentation Part is done by Muskan

MiniLang-Compiler

Project

COURSE

COMPLIER CONSTRCTION

MUSKAN’S CMS ID

023-22-0237

SUHAIL’S CMS ID

023-22-0279

Table of Contents

1. Language Design

1.1 Language Purpose

MiniLang is a custom imperative programming language designed for educational purposes to demonstrate the fundamental phases of compiler construction. It is a strongly-typed language that supports basic arithmetic operations, boolean logic, and essential control flow structures. The language serves as a platform to implement and visualize lexical analysis, syntax parsing, semantic type checking, and intermediate code generation.

1.2 Language Specification

A. **Data Types** MiniLang supports three primitive data types:
- integer: Whole numbers for counting and indexing.
- real: Floating-point numbers for precise calculations.
- boolean: Logical values (true, false) for control flow decisions.

B. **Keywords** The language reserves the following words for structural and logical operations:

program	**var**	**begin**
end	if	then
else	while	do
read	write	integer
real	boolean	and
or	not	true and false

C. **Operators**

Arithmetic: + (Addition), - (Subtraction), * (Multiplication), / (Division)
Relational: = (Equal), <> (Not Equal), < (Less Than), > (Greater Than), <= (Less/Equal), >= (Greater/Equal)
Logical: and, or, not
Assignment: :=

d. Statements The language supports the following executable statements:

- **Assignment:** Assigns an expression result to a variable (e.g., x := 5 + y).
- **Conditional:** Executes code blocks based on boolean conditions (if-then-else).
- **Iterative:** Repeats a block of code while a condition is true (while-do).
- **Input/Output:** Reads values from the user (read) and prints results to the screen (write).

1.3 Informal Grammar Description

A MiniLang program consists of a header defined by program followed by an identifier. This is followed by an optional declaration section (var) where variables and their types are defined. The main execution block is enclosed between begin and end.

Declarations: A list of variables (separated by commas) followed by a colon and their type (e.g., x, y: integer).

Block Structure: Statements within the main block or control structures are separated by semicolons.

Control Flow:

- The if statement evaluates an expression; if true, it executes the then block. An optional else block can follow.
- The while loop repeatedly executes its body as long as the condition remains true.

Expressions: Expressions can be simple values (numbers, identifiers) or complex combinations using arithmetic and relational operators. Operator precedence is enforced (e.g., multiplication before addition).

2. Context-Free Grammar (CFG)

2.1 Original Grammar

Program	→	'program' ID ';' Declarations 'begin' Statements 'end' '.'
Declarations	→	'var' DeclarationList | ε
DeclarationList	→	Declaration ';' DeclarationList | Declaration
Declaration	→	IDList ':' Type
IDList	→	ID ',' IDList | ID
Type	→	'integer' | 'real' | 'boolean'
Statements	→	Statement ';' Statements | Statement
Statement	→	Assignment | IfStmt | WhileStmt | ReadStmt | WriteStmt
Assignment	→	ID ':=' Expression
IfStmt	→	'if' Expression 'then' Statements ElsePart 'end'
ElsePart	→	'else' Statements | ε
WhileStmt	→	'while' Expression 'do' Statements 'end'
ReadStmt	→	'read' '(' ID ')'
WriteStmt	→	'write' '(' Expression ')'
Expression	→	SimpleExpr | SimpleExpr RelOp SimpleExpr
SimpleExpr	→	Term | SimpleExpr AddOp Term
Term	→	Factor | Term MulOp Factor
Factor	→	ID | NUMBER | '(' Expression ')' | 'not' Factor | BOOLEAN_LIT

2.2 Grammar Transformations

To make the grammar compatible with a Recursive Descent Parser (LL(1)), we applied the following mandatory transformations:

A. **Ambiguity Removal (Precedence Enforcement)**

- **Justification:** Standard expression grammars (like E -> E + E) are ambiguous because they don't specify order of operations.
- **Transformation:** We enforced precedence by layering the grammar rules. Factor (parentheses/literals) has the highest precedence, followed by Term

(multiplication/division), and then SimpleExpr (addition/subtraction). This ensures 2 + 3 * 4 is parsed as 2 + (3 * 4) rather than (2 + 3) * 4.

B. **Left Recursion Removal**

- **Justification:** Top-down parsers enter an infinite loop if a rule calls itself as the first step (e.g., Term → Term MulOp Factor). We must convert this "Direct Left Recursion" into "Right Recursion".
- **Transformation: Original:** SimpleExpr → SimpleExpr AddOp Term | Term

Transformed:

SimpleExpr → Term SimpleExpr'
SimpleExpr' → AddOp Term SimpleExpr' | ε

Original: Term → Term MulOp Factor | Factor

Transformed:

Term → Factor Term'
Term' → MulOp Factor Term' | ε

C. **Left Factoring (Non-determinism Removal)**

Justification: The parser cannot decide which rule to choose if two alternatives start with the same symbol (e.g., Expression starts with SimpleExpr in both options). We factor out the common prefix so the decision is delayed until the next unique token is seen.

Transformation:

Original: Expression → SimpleExpr | SimpleExpr RelOp SimpleExpr

Transformed:

Expression → SimpleExpr Expr'
Expr' → RelOp SimpleExpr | ε

Original: Statements → Statement ';' Statements | Statement

Transformed:

Statements → Statement Statements'
Statements' → ';' Statements | ε

2.3 Final Transformed CFG (LL(1))

1. Program	→	'program' ID ';' Declarations 'begin' Statements 'end' '.'
2. Declarations	→	'var' DeclarationList | ε
3. DeclarationList	→	Declaration DList'
4. DList'	→	';' DeclarationList | ε
5. Declaration	→	IDList ':' Type
6. IDList	→	ID IDList'
7. IDList'	→	',' IDList | ε
8. Type	→	'integer' | 'real' | 'boolean'
9. Statements	→	Statement Statements'
10. Statements'	→	';' Statements | ε
11. Statement	→	ID ':=' Expression | 'if' Expression 'then' Statements ElsePart 'end' | 'while' Expression 'do' Statements 'end' | 'read' '(' ID ')' | 'write' '(' Expression ')'
12. ElsePart	→	'else' Statements | ε
13. Expression	→	SimpleExpr Expr'
14. Expr'	→	RelOp SimpleExpr | ε
15. SimpleExpr	→	Term SimpleExpr'
16. SimpleExpr'	→	AddOp Term SimpleExpr' | ε
17. Term	→	Factor Term'

18. Term'	→	MulOp Factor Term' | ε
19. Factor	→	ID | NUMBER | '(' Expression ')' | 'not' Factor | BOOLEAN_LIT
20. RelOp	→	'=' | '<>' | '<' | '>' | '<=' | '>='
21. AddOp	→	'+' | '-' | 'or'
22. MulOp	→	'*' | '/' | 'and'

This is the final grammar implemented in the parser. It is free of left recursion and ambiguity.

3. Lexical Analysis

3.1 Lexer Specifications & Token Definitions

The Lexical Analyzer is implemented in the Lexer class (found in src/lexer.py). It defines the allowed vocabulary of the language using the TokenType class.

A. Token Categories The lexer recognizes the following token patterns:

- **Keywords:** Reserved words that define language structure.

Category	Token Types	Examples
Keywords	PROGRAM, VAR, BEGIN, END, IF, THEN, ELSE, WHILE, DO, READ, WRITE, INTEGER, REAL, BOOLEAN, TRUE, FALSE, AND, OR, NOT	program, if, while
Identifiers	ID	x, fact, myVar
Literals	NUMBER, BOOLEAN_LIT	10, 3.14, true
Operators	PLUS, MINUS, MULT, DIV, ASSIGN, EQ, NEQ, LT, GT, LTE, GTE	+, -, :=, <>
Separators	SEMI, COLON, COMMA, DOT, LPAREN, RPAREN	; , :, ,
Special	EOF	End of file

3.2 Implementation Details

The lexer operates as a state machine that scans the input text one character at a time.

1. Main Loop (get_next_token) The core method get_next_token() continuously advances through the input string. It uses a lookahead character (current_char) to decide how to proceed:

- **Whitespace:** Skips spaces, tabs, and newlines via skip_whitespace().
- **Comments:** Detects {and consumes characters until} is found via skip_comment().
- **Alphanumeric:** If a letter is found, it calls identifier() to consume the full word and check if it is a reserved **Keyword** or a user-defined **ID**.
- **Numeric:** If a digit is found, it calls number() to consume the sequence. It detects the presence of a . to distinguish between INTEGER and REAL tokens.
- **Symbols:** Detects operators. For multi-character operators like := or <=, it "peeks" at the next character to correctly classify the token (e.g., distinguishing < from <>).

2. Regex Logic: Although implemented via direct character scanning for efficiency and precise error reporting, the logic corresponds to standard Regular Expressions:

- **Identifier:** r'[a-z, A-Z_][a-z, A-Z, 0-9_]*`
- **Real Number:** r'\d+\.\d+'
- **Integer:** r'\d+'
- **Comment:** r'\{.*?\}'

3.3 Error Handling & Line Tracking

The lexer maintains two state variables, self.line and self.column, to track the current position in the source file.

- **Line Counting:** Every time a newline character \n is consumed, self.line is incremented, and self.column is reset.
- **Error Reporting:** If the lexer encounters a character that does not match any valid pattern, the error() method is triggered. This raises an exception with the exact location, e.g., "Lexical error at line 5, column 12: Unexpected character '$'".

3.4 Token Stream Example

Input Source:

```
var x: integer;
x := 10;
```

Generated Token Stream:

Token(VAR, 'var', line=1, col=1)
Token(ID, 'x', line=1, col=5)
Token(COLON, ':', line=1, col=6)
Token(INTEGER, 'integer', line=1, col=8)
Token(SEMI, ';', line=1, col=15)
Token(ID, 'x', line=2, col=1)
Token(ASSIGN, ':=', line=2, col=3)
Token(NUMBER, 10, line=2, col=6)
Token(SEMI, ';', line=2, col=8)
Token(EOF, None, line=3, col=1)

4. Parser Selection and Implementation

4.1 Parser Selection

Selected Technique: Recursive Descent Parser (LL(1))

Justification:

1. **Grammar Suitability:** We successfully transformed the language grammar into an **LL(1)** form by eliminating left recursion and applying left factoring. This ensures that for every non-terminal, the parser can decide which production rule to apply by looking at just the first token of the input (Lookahead k=1).
2. **Implementation Simplicity:** Recursive Descent maps every non-terminal in the grammar directly to a function in the code (e.g., program(), statement()). This direct mapping makes the parser easy to write, debug, and maintain compared to table-driven (LR) parsers.
3. **Error Reporting:** Since the parser manages the control flow via the call stack, it preserves the context of where an error occurs, allowing for precise and meaningful error messages (e.g., "Missing semicolon in variable declaration").

4.2 Mathematical Analysis (LL(1) Construction)

To prove the grammar is parsable via LL(1) Recursive Descent, we constructed the **First** and **Follow** sets and the Parsing Table.

First and Follow Sets

These sets determine the decision-making logic of the parser.

Non-Terminal	First Set	Follow Set
Program	{ program }	{ $ }
Declarations	{ var, ε }	{ begin }
DList	{ ID }	{ begin }
Type	{ integer, real, boolean }	{ ; }
Statements	{ ID, if, while, read, write }	{ end, else }
Statements'	{ ;, ε }	{ end, else }
Statement	{ ID, if, while, read, write }	{ ;, end, else }
Expression	{ ID, NUMBER, (, not, true, false }	{ ;, then, do,), end, else }

Term	{ ID, NUMBER, (, not, true, false }	{ +, -, or, ;, then, do,), end, else }
Factor	{ ID, NUMBER, (, not, true, false }	{ *, /, and, +, -, or, ;, then, do,), end, else }

LL(1) Parsing Table

This table maps Non-Terminals (Rows) and Input Tokens (Columns) to specific Grammar Rules. (Note: Empty cells indicate a Syntax Error. ε indicates the epsilon/empty production).

NT\ Token	ID	if	while	var	;	begin	end	$
Program								
Rule								
Decls				var DList		ε		
Stmts	Stmt Stmts'	Stmt Stmts'	Stmt Stmts'					
Stmts'					; Stmts		ε	
Stmt	Assign	IfStmt	WhileStmt					

Explanation:

- When the parser is at **Decls** and sees var, it chooses the rule var DList.
- When the parser is at **Decls** and sees begin (which is in the Follow set), it applies the epsilon rule ε (meaning no variables are declared).
- When the parser is at **Stmts** and sees if, it calls IfStmt.

4.3 Implementation Details

The parser is implemented as a class Parser in src/parser.py. It works in conjunction with the Lexer to build the Abstract Syntax Tree (AST).

Key Components

1. **Lookahead (current_token):** The parser maintains a reference to the current token from the Lexer to decide which method to call next.

2. **eat(token_type) Method:**

    - Verifies that the current_token matches the expected token_type.
    - If it matches, it consumes the token and advances to the next one.
    - If it does not match, it triggers the Error Handler.

3. **Parsing Methods:** Each grammar rule has a corresponding method. For example, the while_stmt method:

```
def while_stmt(self):
    self.eat(TokenType.WHILE)   # Match 'while'
    expr = self.expression()    # Parse condition
    self.eat(TokenType.DO)      # Match 'do'
    body = self.statements()    # Parse loop body
    self.eat(TokenType.END)     # Match 'end'
    return WhileNode(expr, body)
```

4.4 Error Handling & Recovery

A. Error Detection

The parser detects syntax errors when the current token does not match the grammar expectation.

- **Mechanism:** The error(message) method is called, which raises a SyntaxError exception.
- **Reporting:** The error message includes the **Line Number**, **Column Number**, and a description of the mismatch (e.g., "*Expected 'THEN' but found 'ID' at line 5*").

B. Error Recovery (Panic Mode)

To prevent the compiler from crashing on the first error, we implement a **Panic Mode** recovery strategy.

- **Strategy:** When an error occurs inside a statement or declaration, the parser enters "recovery mode." It discards tokens until it finds a **Synchronization Token** (safe delimiter) such as ; (semicolon) or end.
- **Benefit:** This allows the compiler to continue checking the rest of the file for other errors, rather than stopping immediately.

```
def recover(self):

    # Skip tokens until a semi-colon or end is found

    while self.current_token.type not in [TokenType.SEMI, TokenType.END,
TokenType.EOF]:

        self.advance()

    if self.current_token.type == TokenType.SEMI:

        self.advance() # Consume the semi-colon to start fresh
```

5. Semantic Rules and Actions

5.1 Overview

The semantic analysis phase ensures that the syntactically correct program follows the logical rules of the language. This is implemented in the SemanticAnalyzer class, which traverses the Abstract Syntax Tree (AST) to validate variable scope and type consistency.

5.2 Symbol Table Management

The **Symbol** Table is the central data structure used to track variable declarations and their associated types.

- **Implementation:** A simple Python dictionary (self.symbol_table) maps variable names (strings) to their type names (strings like 'integer', 'real', 'boolean').
- **Action (Declaration):** When the analyzer visits a DeclarationNode (produced by var x, y: integer ;), it iterates through the identifiers and registers them in the table.

```
def analyze_declaration(self, node):
    for identifier in node.identifiers:
        self.symbol_table[identifier] = node.type_node.type_name
```

- **Action (Lookup):** When an assignment or expression uses a variable (VarNode or ReadNode), the analyzer checks if the name exists in the symbol_table. If not, a semantic error is raised.

5.3 Type System & Type Checking

The compiler implements a Static Type System enforced by the TypeChecker class. It defines how types interact during operations and assignments.

A. Binary Operation Rules

The check_binary_operation method enforces the following matrix:

Operator Category	Operators	Operand Types	Result Type	Rule
Arithmetic	+, -, *, /	integer, real	integer or real	If both are integer, result is integer. If at least one is real, result is real (implicit promotion).
Relational	=, <>, <, >, <=, >=	Same Type OR Numeric	boolean	Operands must be the same type, or both numeric (int/real).
Logical	and, or	boolean	boolean	Both operands must strictly be boolean.

B. Unary Operation Rules

- **Operator:** not

- **Rule:** Can only be applied to an expression of type boolean. Any other type results in a Type error.

C. Assignment Rules

Variables are strictly typed. The type of the expression on the right-hand side (RHS) must be compatible with the variable on the left-hand side (LHS).

- **Strict Match:** boolean variables can only be assigned boolean values.
- **Numeric Compatibility:** An integer value can be assigned to an integer variable.
- ***Note:*** The system prevents assigning incompatible types, e.g., assigning a boolean to an integer variable raises an exception.

D. Control Flow Rules

For IfNode and WhileNode, the condition expression **must** evaluate to type boolean.

- **Check:** if condition_type != 'boolean': raise Exception(...).

5.4 Semantic Error Reporting

The semantic analyzer detects logical errors that the parser cannot catch. Errors are reported immediately by raising exceptions with descriptive messages.

Implemented Semantic Errors:

1. **Undeclared Variables:**
    - Trigger: Using a variable in an assignment, expression, or read statement without declaring it in the var block.
    - Message: "Semantic error: Variable 'x' not declared".
2. **Type Mismatch in Assignments:**
    - Trigger: x := true where x is declared as integer.
    - Message: "Type error: Cannot assign boolean to integer variable 'x' ".

3. **Incompatible Binary Operators:**

    - Trigger: 10 + true (Adding a number to a boolean).
    - Message: "Type error: Incompatible types integer and boolean for operator +".

4. **Invalid Control Flow Conditions:**

    - Trigger: if 5 + 10 then ... (Condition is a number, not a boolean).
    - Message: "Type error: If condition must be boolean, got integer".

6. Intermediate Code Generation

6.1 Methodology: Syntax-Directed Translation

The Intermediate Code Generation (ICG) phase transforms the high-level Abstract Syntax Tree (AST) into a machine-independent intermediate representation. We utilize **Three-Address Code (TAC)**, a sequence of instructions where each instruction has at most three operands (e.g., result = operand1 op operand2).

Why TAC?

- **Simplicity:** It linearizes complex expressions (like a + b * c) into simple steps that are closer to assembly language.
- **Optimization:** It is easier to perform optimizations (like common subexpression elimination) on TAC than on the AST.
- **Portability:** The TAC can be easily translated into assembly code for various target machines.

6.2 Implementation Details

The ICG is implemented in the TACGenerator class (src/tac.py). It traverses the AST and appends instructions to a linear list.

A. Instruction Format

Instructions are represented as tuples or strings in the format: (Operator, Arg1, Arg2, and Result). Common Opcodes implemented include:

- **Arithmetic:** ADD, SUB, MUL, DIV
- **Relational:** LT (<), GT (>), EQ (==), etc.

- **Logic/Control:** IF_FALSE (conditional jump), GOTO (unconditional jump), LABEL (jump target).
- **Data Movement:** COPY (assignment).
- **I/O:** READ, WRITE.

B. Temporary Variable & Label Management

To manage intermediate results and control flow, the generator dynamically creates unique identifiers.

Temporary Variables (t0, t1 ...): Generated by the new_temp() method. Used to store the results of sub-expressions.

```
def new_temp(self):
    name = f"t{self.temp_count}"
    self.temp_count += 1
    return name
```

[cite: uploaded:MiniLang-Compiler/src/tac.py]

Labels (L1, L2, ...): Generated by the new_label() method. Used as targets for GOTO and IF_FALSE instructions to implement if-then-else and while loops.

```
def new_label(self):
    name = f"L{self.label_count}"
    self.label_count += 1
    return name
```

[cite: uploaded:MiniLang-Compiler/src/tac.py]

6.3 Translation Schemes

The following schemes illustrate how high-level constructs are translated into TAC.

1. Binary Expressions (A + B)

To translate E1 op E2:

1. Generate code for E1 -> store result in addr1.
2. Generate code for E2 -> store result in addr2.
3. Create new temp t.
4. Emit instruction: t = addr1 op addr2.

2. Assignments (X := E)

To translate id := E:

1. Generate code for E -> store result in addr.
2. Emit instruction: id = addr.

3. If-Else Statements

Structure: if E then S1 else S2

1. Generate code for condition E -> result in t_cond.
2. Create labels L_else and L_end.
3. Emit IF_FALSE t_cond GOTO L_else.
4. Generate code for S1 (Then block).
5. Emit GOTO L_end.
6. Emit LABEL L_else.
7. Generate code for S2 (Else block).
8. Emit LABEL L_end.

4. While Loops

Structure: while E do S

1. Create labels L_start and L_end.
2. Emit LABEL L_start.
3. Generate code for condition E -> result in t_cond.
4. Emit IF_FALSE t_cond GOTO L_end.
5. Generate code for body S.
6. Emit GOTO L_start.
7. Emit LABEL L_end.

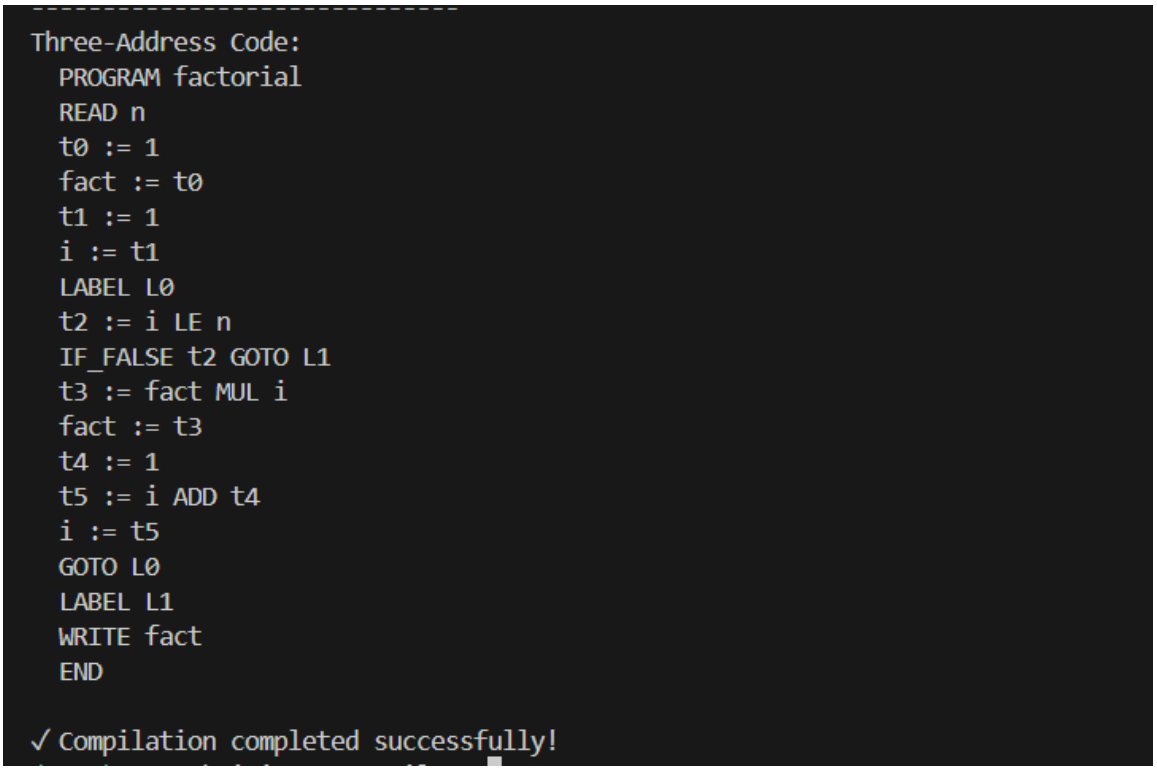
6.4 Example Input & Generated Code

Input Program (`example2.ml`): This program calculates the factorial of 5.

examples > example2.ml

```
1   program factorial;
2   var
3     n, i, fact: integer;
4   begin
5     read(n);
6     fact := 1;
7     i := 1;
8     while i <= n do
9     begin
10      fact := fact * i;
11      i := i + 1;
12    end;
13    write(fact);
14  end.
15
```

Generated Intermediate Code (TAC): The compiler output demonstrates correct label placement for the loop and temp usage for the multiplication and addition.



7. Test Cases & Results

Example 1:

compiler.py example1.ml

examples > example1.ml

```
1   program example1;
2   var
3     x, y: integer;
4     max: integer;
5   begin
6     read(x);
7     read(y);
8     if x > y then
9       max := x  { no semicolon needed before else }
10    else
11      max := y; { semicolon needed if followed by another statement }
12    write(max);
13  end.
```

Output:

```
(venv) PS E:\MiniLang-Compiler> python src/compiler.py examples/example1.ml
=== MiniLang Compiler ===
Source code:
program example1;
var
  x, y: integer;
  max: integer;
begin
  read(x);
  read(y);
  if x > y then
    max := x  { no semicolon needed before else }
  else
    max := y; { semicolon needed if followed by another statement }
  write(max);
end.

==============================================
```

```
1. LEXICAL ANALYSIS:
------------------------------
Token(PROGRAM, 'program', line=1, col=8)
Token(ID, 'example1', line=1, col=17)
Token(SEMI, ';', line=1, col=18)
Token(VAR, 'var', line=2, col=4)
Token(ID, 'x', line=3, col=4)
Token(COMMA, ',', line=3, col=5)
Token(ID, 'y', line=3, col=7)
Token(COLON, ':', line=3, col=8)
Token(INTEGER, 'integer', line=3, col=16)
Token(SEMI, ';', line=3, col=17)
Token(ID, 'max', line=4, col=6)
Token(COLON, ':', line=4, col=7)
Token(INTEGER, 'integer', line=4, col=15)
Token(SEMI, ';', line=4, col=16)
Token(BEGIN, 'begin', line=5, col=6)
Token(READ, 'read', line=6, col=7)
Token(LPAREN, '(', line=6, col=8)
Token(ID, 'x', line=6, col=9)
Token(RPAREN, ')', line=6, col=10)
Token(SEMI, ';', line=6, col=11)
Token(READ, 'read', line=7, col=7)
Token(LPAREN, '(', line=7, col=8)
Token(ID, 'y', line=7, col=9)
Token(RPAREN, ')', line=7, col=10)
Token(SEMI, ';', line=7, col=11)
Token(IF, 'if', line=8, col=5)
Token(ID, 'x', line=8, col=7)
Token(GT, '>', line=8, col=9)
Token(ID, 'y', line=8, col=11)
Token(THEN, 'then', line=8, col=16)
Token(ID, 'max', line=9, col=8)
Token(ASSIGN, ':=', line=9, col=11)
Token(ID, 'x', line=9, col=13)
Token(ELSE, 'else', line=10, col=7)
Token(ID, 'max', line=11, col=8)
Token(ASSIGN, ':=', line=11, col=11)
Token(ID, 'y', line=11, col=13)
Token(SEMI, ';', line=11, col=14)
Token(WRITE, 'write', line=12, col=8)
Token(LPAREN, '(', line=12, col=9)
Token(ID, 'max', line=12, col=12)
Token(RPAREN, ')', line=12, col=13)
Token(SEMI, ';', line=12, col=14)
Token(END, 'end', line=13, col=4)
Token(DOT, '.', line=13, col=5)
Token(EOF, 'None', line=13, col=5)
Total tokens: 46
```

```
2. SYNTAX ANALYSIS:
------------------------------
✓ Parsing completed successfully
✓ Symbol table: 3 variables declared
✓ Parsing successful!
Program: example1
Declarations: 2
Statements: 4

AST Details:
  Declaration 1: Declaration(['x', 'y']: Type(integer))
  Declaration 2: Declaration(['max']: Type(integer))
  Statement 1: Read(x)
  Statement 2: Read(y)
  Statement 3: If(BinOp(Var(x) > Var(y)))
  Statement 4: Write(Var(max))

3. SEMANTIC ANALYSIS:
------------------------------
✓ Semantic analysis successful!
✓ All variables properly declared
✓ Type checking passed

4. INTERMEDIATE CODE GENERATION:
------------------------------
Three-Address Code:
  PROGRAM example1
  READ x
  READ y
  t0 := x GT y
  IF_FALSE t0 GOTO L0
  max := x
  GOTO L1
  LABEL L0
  max := y
  LABEL L1
  WRITE max
  END

✓ Compilation completed successfully!
(venv) PS E:\MiniLang-Compiler>
```

Test Case 2: Syntax Error Handling

compiler.py | example5.ml

examples > example5.ml

```
1   program error1;
2   var x: integer;
3   begin
4     x = 10;  { Error: Used '=' instead of ':=' }
5   end.
```

Output

```
(venv) PS E:\MiniLang-Compiler> python src/compiler.py examples/example5.ml
=== MiniLang Compiler ===
Source code:
program error1;
var x: integer;
begin
  x = 10;  { Error: Used '=' instead of ':=' }
end.

================================================

1. LEXICAL ANALYSIS:
------------------------------
Token(PROGRAM, 'program', line=1, col=8)
Token(ID, 'error1', line=1, col=15)
Token(SEMI, ';', line=1, col=16)
Token(VAR, 'var', line=2, col=4)
Token(ID, 'x', line=2, col=6)
Token(COLON, ':', line=2, col=7)
Token(INTEGER, 'integer', line=2, col=15)
Token(SEMI, ';', line=2, col=16)
Token(BEGIN, 'begin', line=3, col=6)
Token(ID, 'x', line=4, col=4)
Token(EQ, '=', line=4, col=6)
Token(ID, 'x', line=4, col=4)
Token(EQ, '=', line=4, col=6)
Token(NUMBER, '10', line=4, col=9)
Token(SEMI, ';', line=4, col=10)
Token(END, 'end', line=5, col=4)
Token(DOT, '.', line=5, col=5)
Token(EOF, 'None', line=5, col=5)
Total tokens: 16

2. SYNTAX ANALYSIS:
------------------------------

✗ Compilation error: Parsing failed: Syntax error at line 4: Expected 'ASSIGN', but found 'EQ' at line 4
Current token: Token(EQ, '=', line=4, col=6)
```

Test Case 3: Semantic Error Handling

compiler.py | example5.ml | example6.ml

examples > example6.ml

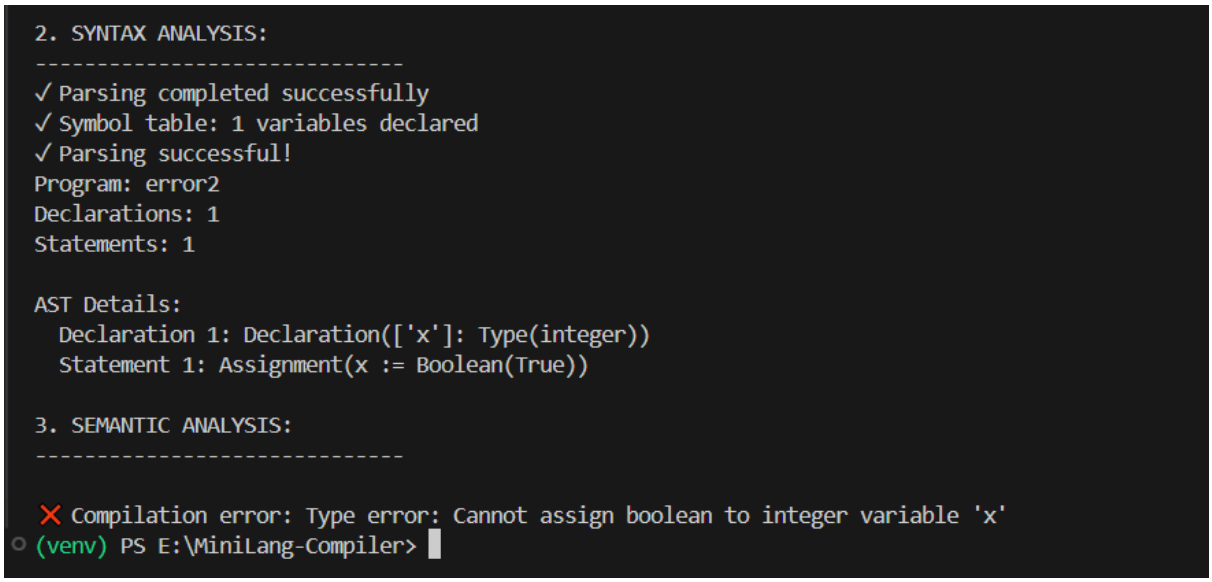
```
1  program error2;
2  var x: integer;
3  begin
4    x := true;  { Error: Assigning boolean to integer }
5  end.
```

Output

```
(venv) PS E:\MiniLang-Compiler> python src/compiler.py examples/example6.ml
=== MiniLang Compiler ===
Source code:
program error2;
var x: integer;
begin
  x := true;  { Error: Assigning boolean to integer }
end.

================================================

1. LEXICAL ANALYSIS:
----------------------------
Token(PROGRAM, 'program', line=1, col=8)
Token(ID, 'error2', line=1, col=15)
Token(SEMI, ';', line=1, col=16)
Token(VAR, 'var', line=2, col=4)
Token(ID, 'x', line=2, col=6)
Token(COLON, ':', line=2, col=7)
Token(INTEGER, 'integer', line=2, col=15)
Token(SEMI, ';', line=2, col=16)
Token(BEGIN, 'begin', line=3, col=6)
Token(ID, 'x', line=4, col=4)
Token(ASSIGN, ':=', line=4, col=7)
Token(BOOLEAN_LIT, 'true', line=4, col=12)
Token(SEMI, ';', line=4, col=13)
Token(END, 'end', line=5, col=4)
Token(DOT, '.', line=5, col=5)
Token(EOF, 'None', line=5, col=5)
Total tokens: 16
```


2. SYNTAX ANALYSIS:

✓ Parsing completed successfully
✓ Symbol table: 1 variables declared
✓ Parsing successful!
Program: error2
Declarations: 1
Statements: 1
AST Details:
Declaration 1: Declaration(['x']: Type(integer))
Statement 1: Assignment(x := Boolean(True))
3. SEMANTIC ANALYSIS:

❌ Compilation error: Type error: Cannot assign boolean to integer variable 'x'
(venv) PS E:\MiniLang-Compiler>