# **MiniPascal Compiler - Project Status Report**

Date of Report: June 4, 2025

Project Lead: Tarek

## **1. Introduction & Project Goal**

This report details the current status of the MiniPascal Compiler project. The primary objective is to develop a compiler for "MiniPascal," a custom, simplified Pascal-like programming language. This involves several key phases, and this report focuses on the significant progress made in the initial **Lexical Analysis** and **Syntax Analysis** stages. The compiler is being developed using Flex for lexical analysis, Bison for syntax analysis (parser generation), and C++ for implementing the Abstract Syntax Tree (AST) and associated logic.

The compiler can now successfully parse a substantial subset of the MiniPascal language, transforming source code into a structured Abstract Syntax Tree (AST) representation, which is then printed for verification.

## **2. Core Technologies & Tools**

* **Lexical Analysis (Scanner):** GNU Flex (generates scanner.cpp from lexer.l)
* **Syntax Analysis (Parser):** GNU Bison (generates parser.cpp and parser.h from parser.y)
* **AST & Supporting Logic:** C++ (std=c++17)
* **Build System:** GNU Make with g++ compiler
* **Development Environment (deduced):** MinGW64 on Windows

## **3. Lexical Analysis (The Lexer)**

The lexer is responsible for reading the MiniPascal source code and breaking it down into a stream of tokens – the basic building blocks of the language.

**Key Capabilities:**

* **Whitespace and Line Tracking:**
  + Correctly ignores spaces and tabs.
  + Accurately tracks current line and column numbers (1-indexed) for error reporting and AST node location.
* **Comments:**
  + Handles single-line comments: // ... until end of line
  + Handles multi-line comments: { ... }, including detection of unterminated multi-line comments.
* Keywords: Recognizes the following MiniPascal keywords (case-insensitively):  
  PROGRAM, VAR, INTEGER, REAL, BOOLEAN, ARRAY, OF, FUNCTION, PROCEDURE, BEGIN, END, IF, THEN, ELSE, WHILE, DO, DIV, NOT, OR, AND, TRUE, FALSE, RETURN.
* **Identifiers:**
  + Recognizes user-defined names for variables, procedures, functions, etc.
  + Rules: Must start with a letter or underscore, followed by any number of letters, digits, or underscores (e.g., myVar, \_count1, Process\_Data).
* **Literals:**
  + **Integer Literals:** Sequences of digits (e.g., 123, 0, 42).
  + **Real Literals:** Numbers with a decimal point and/or an exponent. The lexer has been specifically refined to correctly distinguish real numbers from integer-dot-dot sequences (e.g., 1..5).
    - Examples: 1.0, 0.5, 123.456, 1.2e3, .5E-2, 10e+0.
    - Importantly, 1. by itself is *not* treated as a real literal (it's tokenized as NUM(1) then .) unless followed by more digits or an exponent, ensuring ARRAY [1..5] parses correctly.
  + **String Literals:** Sequences of characters enclosed in single quotes (e.g., 'Hello, World!', 'A'). Supports basic escape sequences like \'.
  + **Boolean Literals:** TRUE, FALSE (tokenized as TRUE\_KEYWORD, FALSE\_KEYWORD).
* **Operators & Punctuation:** Recognizes a comprehensive set:
  + Assignment: :=
  + Relational: =, <>, <, <=, >, >=
  + Arithmetic: +, -, \*, /, DIV (integer division)
  + Logical: AND, OR, NOT
  + Structural: (, ), [, ], :, ;, ,, .
  + Array Range: .. (dot-dot)

**Lexer Output:** For each recognized element, the lexer provides the parser with a token type and, where applicable, the actual value (e.g., the number for NUM, the string for IDENT or STRING\_LITERAL).

## **4. Syntax Analysis (The Parser & Grammar)**

The parser takes the stream of tokens from the lexer and verifies if they form a grammatically correct MiniPascal program according to the language's syntax rules. If the program is valid, the parser constructs an Abstract Syntax Tree (AST).

**MiniPascal Language Grammar Highlights:**

### **4.1. Overall Program Structure**

A MiniPascal program has the following top-level structure:

PROGRAM <program\_name>;  
 <global\_declarations>  
 <subprogram\_declarations>  
BEGIN // Main program body  
 <statement\_sequence>  
END.

* The program starts with the PROGRAM keyword, a name, and a semicolon.
* Global declarations (variables) are optional.
* Subprogram (procedure/function) declarations are optional.
* The main executable part is a compound statement (BEGIN...END).
* The entire program must end with a period (.).

### **4.2. Declarations**

* **Variable Declarations (VAR section):**
  + Syntax: VAR <identifier\_list> : <type>; ...
  + Multiple variables of the same type can be declared in a list (e.g., x, y : Integer;).
  + Multiple VAR items can follow each other.
  + Supported in both the global scope and locally within procedures and functions.
* **Type System:**
  + **Standard Types:**
    - INTEGER
    - REAL
    - BOOLEAN
  + **Array Types:**
    - Syntax: ARRAY [ <low\_bound\_int> .. <high\_bound\_int> ] OF <standard\_type>;
    - Example: myArray : ARRAY [1..10] OF INTEGER;
    - Array bounds must be integer literals.
    - Arrays can only be of standard types.

### **4.3. Subprogram Declarations (Procedures and Functions)**

* **Procedure Declarations:**
  + Syntax: PROCEDURE <proc\_name> ( <parameter\_list> ); <local\_declarations> BEGIN <statement\_sequence> END;
  + Parameters are optional. If present, they are enclosed in parentheses.
  + Local variable declarations (VAR section) are optional and appear before the procedure's BEGIN.
  + Example:  
    PROCEDURE Display(message : STRING\_LITERAL\_TYPE\_placeholder; value : INTEGER); // STRING\_LITERAL\_TYPE\_placeholder indicates string literals are passed  
     VAR temp : INTEGER;  
    BEGIN  
     // statements  
    END;
* **Function Declarations:**
  + Syntax: FUNCTION <func\_name> ( <parameter\_list> ) : <return\_type>; <local\_declarations> BEGIN <statement\_sequence> END;
  + Similar to procedures but include a colon and a return\_type (must be a standard\_type).
  + Must contain at least one RETURN <expression>; statement (parsing-wise; semantic check for this is future work).
  + Example:  
    FUNCTION Add(a : INTEGER; b : INTEGER) : INTEGER;  
     VAR sum\_local : INTEGER;  
    BEGIN  
     sum\_local := a + b;  
     RETURN sum\_local;  
    END;
* **Parameters:**
  + Syntax: <identifier\_list> : <type>
  + Multiple parameter groups are separated by semicolons (e.g., (a,b : INTEGER; flag : BOOLEAN)).
  + Currently, parameters are passed by value (conceptually, as MiniPascal doesn't have explicit by-reference syntax yet).

### **4.4. Statements**

The executable parts of a MiniPascal program are composed of statements:

* **Assignment Statement:**
  + Syntax: <variable\_access> := <expression>;
  + Examples: myVar := 10;, myArray[i] := anotherVar + 5;
* **Compound Statement:**
  + Syntax: BEGIN <statement\_sequence> END
  + Allows grouping multiple statements.
  + Trailing semicolons before END are optional for the last statement in the list.
* **Procedure Call Statement:**
  + Syntax: <procedure\_name> ( <argument\_list> ); or <procedure\_name>; (for parameterless).
  + Arguments are expressions.
  + Examples: DisplayMessage('Hello');, Calculate(x, 5\*y);, NewLine;
* **IF Statement:**
  + Syntax: IF <condition\_expr> THEN <statement>
  + Syntax: IF <condition\_expr> THEN <statement> ELSE <statement>
  + Handles the "dangling else" ambiguity by associating ELSE with the nearest unmatched IF.
* **WHILE Statement:**
  + Syntax: WHILE <condition\_expr> DO <statement>
* **RETURN Statement (for Functions):**
  + Syntax: RETURN <expression>;
  + Used within functions to specify the value to be returned.

### **4.5. Expressions**

MiniPascal supports arithmetic, relational, and logical expressions with operator precedence and associativity rules:

* **Operators (by decreasing precedence):**
  1. UMINUS (unary minus, e.g., -x) - Highest
  2. \*, /, DIV (multiplicative, left-associative)
  3. +, - (additive, left-associative)
  4. =, <>, <, <=, >, >= (relational, non-associative)
  5. NOT (logical negation)
  6. AND (logical and, left-associative)
  7. OR (logical or, left-associative) - Lowest
* **Operands:**
  + Literals (integer, real, boolean, string)
  + Variable access (simple identifiers, array element access e.g., myArray[index\_expr])
  + Function calls (e.g., MyFunction(arg1, arg2))
  + Parenthesized expressions: ( ... ) to override precedence.

### **4.6. Variable Access**

* **Simple Variables:** myVariable
* **Array Element Access:** myArrayIdentifier [ <expression> ] (index expression must evaluate to an integer type, checked semantically later).

## **5. Abstract Syntax Tree (AST)**

Upon a successful parse, an Abstract Syntax Tree is constructed. The AST is a hierarchical tree representation of the source code's structure, making it easier for subsequent compiler phases (like semantic analysis and code generation) to process.

* **Structure:** A tree of C++ objects, where each object (node) represents a language construct (e.g., ProgramNode, VarDeclNode, AssignStatementNode, BinaryOpNode).
* **Node Information:** Each AST node stores relevant information, such as line and column numbers, identifiers, literal values, child nodes, etc.
* **Verification:** A print() method is implemented for each AST node type, allowing the entire tree to be printed in an indented, human-readable format. This has been crucial for verifying the correctness of the parser and AST construction.

## **6. MiniPascal Language Capabilities & Limitations**

### **Current Capabilities:**

* **Program Definition:** Full program structure with global variables, procedures, functions, and a main body.
* **Data Types:** INTEGER, REAL, BOOLEAN, and one-dimensional ARRAYs of these standard types with literal integer bounds.
* **Declarations:** VAR declarations for global and local scope (procedures/functions). Parameter declarations.
* **Statements:**
  + Assignment (:=)
  + Compound (BEGIN...END)
  + Procedure calls (with and without arguments)
  + IF...THEN...ELSE (and IF...THEN)
  + WHILE...DO
  + RETURN <expression> (for functions)
* **Expressions:**
  + Arithmetic (+, -, \*, /, DIV)
  + Relational (=, <>, <, <=, >, >=)
  + Logical (AND, OR, NOT)
  + Function calls within expressions.
  + Parentheses for grouping.
  + Correct operator precedence and associativity.
* **Literals:** Integer, Real, Boolean (TRUE/FALSE), String.
* **Comments:** Single-line (//) and multi-line ({}).
* **Basic I/O (Parsing):** Calls to procedures like Read, ReadLn, Write, WriteLn are parsed as standard procedure calls. Their special semantics (variable arguments, type handling) are for later phases.

### **Current Limitations (Not Yet Implemented / Out of Scope for Parsing Phase):**

* **No CONST declarations.**
* **No user-defined TYPE declarations** (e.g., enumerations, subranges, records).
* **No CHAR data type.**
* **No advanced control flow:** FOR loops, REPEAT...UNTIL loops, CASE statements.
* **No explicit pass-by-reference for parameters.**
* **No** modules/units system (beyond parsing PROGRAM <name>;).
* **No pointers or dynamic memory allocation.**
* **Limited String Operations:** Strings are primarily literals; no concatenation or complex string manipulation functions are built-in.
* **Error Recovery:** While basic syntax errors are reported with line/column numbers, sophisticated error recovery within the parser is minimal.
* **Semantic Checks:** This report focuses on parsing. All semantic checks (type compatibility, declaration before use, scope resolution, etc.) are part of the next compiler phase.
* **Code Generation/Interpretation:** The compiler currently only parses and builds an AST. It does not yet execute or translate MiniPascal code.

## **7. Key Successes & Milestones Achieved in Lexing/Parsing**

* **Robust Literal Tokenization:** Successfully distinguished REAL\_LITERALs from sequences like 1.. (integer followed by dot-dot operator).
* **Operator Precedence:** Correctly implemented a cascaded expression grammar to enforce standard operator precedence and associativity.
* **Control Structures:** Accurate parsing of nested IF-THEN-ELSE and WHILE-DO loops.
* **Subprogram Handling:** Successful parsing of PROCEDURE and FUNCTION declarations, including:
  + Parameter lists (0, 1, or multiple parameters of different types).
  + Return types for functions.
  + Local VAR declarations within both procedures and functions.
  + RETURN statements in functions.
* **Array Declarations and Access:** Correct parsing of 1D array declarations and element access (myArray[index\_expr]) on both sides of assignments and in expressions.
* **AST Verification:** The print() functionality of the AST has been invaluable for debugging and confirming the parser's output.

## **8. Illustrative MiniPascal Code Examples (Parsable)**

The following examples demonstrate code that the MiniPascal compiler can currently parse successfully:

**Example** 1: Array **and Basic Statements**

PROGRAM ArrayExample;  
VAR  
 myArray : ARRAY [1..5] OF INTEGER;  
 myVar : INTEGER;  
BEGIN  
 myArray[1] := 10;  
 myVar := myArray[1] + 5;  
 IF myVar > 10 THEN  
 myVar := myVar \* 2  
 ELSE  
 myVar := 0;  
END.

**Example 2: Procedure with Local Variables and Parameters**

PROGRAM ProcExample;  
PROCEDURE CalculateSum(a : INTEGER; b : INTEGER);  
 VAR  
 tempSum : INTEGER;  
 BEGIN  
 tempSum := a + b;  
 // Assuming a WriteLn procedure is known for I/O (parsed as a call)  
 WriteLn('Sum is: ', tempSum);  
 END;  
BEGIN // Main  
 CalculateSum(15, 27);  
END.

**Example 3: Function with Local Variables and Return Statement**

PROGRAM FuncReturnExample;  
VAR  
 globalResult : INTEGER;  
  
FUNCTION ComputeMax(x : INTEGER; y : INTEGER) : INTEGER;  
 VAR  
 maxVal : INTEGER;  
 BEGIN  
 IF x > y THEN  
 maxVal := x  
 ELSE  
 maxVal := y;  
 RETURN maxVal;  
 END;  
  
BEGIN // Main  
 globalResult := ComputeMax(100, 200);  
 // WriteLn('Max is: ', globalResult); // For display  
END.

## **9. Next Steps**

With the lexical and syntax analysis phases largely complete and robust for the defined MiniPascal subset, the immediate next major phase for the project is **Semantic Analysis**. This will involve:

* Building a Symbol Table.
* Performing type checking.
* Verifying declarations and scope rules.
* Annotating the AST with semantic information.
* Implementing the Visitor design pattern to facilitate these checks.