## **MiniPascal Compiler - Project Status Report (Semantic Analysis Phase)**

Date of Report: June 5, 2025

Project Lead: Tarek

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

This report details the current status of the MiniPascal Compiler project, focusing on the successful completion of the **Semantic Analysis phase**. The primary objective of the overall project is to develop a functional compiler for "MiniPascal," a custom, simplified Pascal-like programming language.

Following the successful Lexical and Syntax Analysis stages (which produce an Abstract Syntax Tree - AST), the Semantic Analysis phase is responsible for verifying the meaning and consistency of the code. This involves checking for type errors, scope violations, and ensuring that language rules beyond basic syntax are adhered to. The compiler now successfully performs these checks, annotates the AST with semantic information (like determined types), and reports detailed errors when semantic rules are violated.

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

The core technologies remain consistent, with new components introduced for semantic processing:

* **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)
* **Abstract Syntax Tree (AST) & Supporting Logic:** C++ (std=c++17)
  + AST nodes (ast.h, ast.cpp) now include members to store determined semantic information (e.g., determinedType in ExprNode).
* **Semantic Analysis Engine:**
  + **Symbol Table:** Custom C++ implementation (symbol\_table.h, symbol\_table.cpp) for managing identifiers, their kinds, types, scopes, and parameter signatures.
  + **Visitor Pattern:** Implemented (semantic\_visitor.h, semantic\_analyzer.h, semantic\_analyzer.cpp) to traverse the AST and apply semantic checks.
  + **Type System Definitions:** Enumerations for symbol kinds and type categories (semantic\_types.h).
* **Build System:** GNU Make with g++ compiler
* **Development Environment:** MinGW64 on Windows

### **3. Semantic Analysis**

The Semantic Analysis phase takes the AST generated by the parser and performs a series of checks to ensure the MiniPascal program is meaningful and follows all language rules not enforceable by syntax alone. It enriches the AST with type information and other semantic details.

**Key Capabilities & Checks Implemented:**

**3.1. Symbol Table Management & Scope Resolution:**

* **Symbol Table Implementation:** A robust symbol table manages identifiers across different scopes.
  + Stores symbol name, kind (variable, parameter, procedure, function, program name), type (Integer, Real, Boolean, Array - including element type and bounds), and declaration location.
  + For functions/procedures, it stores their parameter signatures (types and order).
* **Scope Handling:**
  + **Global Scope:** Manages the program name, global variables, and top-level procedure/function declarations.
  + **Subprogram Scope:** Each procedure and function introduces a new nested scope for its parameters and local variables.
  + Lookup logic correctly searches from the current (innermost) scope outwards to the global scope.
* **Declaration Processing:**
  + **Undeclared Identifiers:** All used identifiers (variables, functions, procedures) must be declared before use. The analyzer reports errors for undeclared identifiers.
  + **Redeclaration:** An identifier cannot be re-declared within the same scope. Errors are reported for such conflicts, noting the location of the previous declaration.
  + Built-in procedures (read, readln, write, writeln) are pre-populated into the global symbol table.

**3.2. Type Checking & Type System Enforcement:**

* **AST Annotation:** Expression nodes in the AST (ExprNode and its derivatives) are annotated with their determined semantic type (e.g., PRIMITIVE\_INTEGER, PRIMITIVE\_REAL, PRIMITIVE\_BOOLEAN, ARRAY with details) after analysis.
* **Literal Types:** Correctly identifies the types of integer, real, boolean, and string literals.
* **Variable & Parameter Access:**
  + Determines the type of a variable or parameter based on its declaration in the symbol table.
  + For array access (myArray[index\_expr]):
    - Verifies that the base identifier (myArray) is indeed an array.
    - Verifies that the index\_expr evaluates to an INTEGER.
    - The type of the array access expression is correctly determined as the element type of the array.
* **Expression Type Determination:**
  + **Arithmetic Operators (+, -, \*, /, DIV):**
    - +, -, \*: Operands must be numeric (INTEGER or REAL). Result is REAL if either operand is REAL, otherwise INTEGER.
    - / (Real Division): Operands must be numeric. Result is always REAL.
    - DIV (Integer Division): Operands must both be INTEGER. Result is INTEGER.
    - Type errors are reported for non-numeric operands.
  + **Relational Operators (=, <>, <, <=, >, >=):**
    - Operands must be of compatible types (both numeric, or both BOOLEAN for = and <>).
    - Result is always BOOLEAN.
    - Errors reported for incompatible operand types.
  + **Logical Operators (AND, OR, NOT):**
    - Operands for AND, OR must be BOOLEAN. Result is BOOLEAN.
    - Operand for NOT must be BOOLEAN. Result is BOOLEAN.
    - Errors reported for non-BOOLEAN operands.
  + **Unary Minus (-):** Operand must be numeric (INTEGER or REAL). Result type matches operand type.
* **Assignment Statement (variable := expression):**
  + The type of the expression on the right-hand side must be compatible with the type of the variable on the left-hand side.
  + Compatibility rules:
    - Identical types are compatible.
    - An INTEGER expression can be assigned to a REAL variable.
    - Assigning a REAL expression to an INTEGER variable is an error.
  + Whole array assignment is not supported and flagged as an error.
* **Control Flow Statements:**
  + **IF <condition> THEN ...**: The condition expression must evaluate to BOOLEAN.
  + **WHILE <condition> DO ...**: The condition expression must evaluate to BOOLEAN.
* **Subprogram Calls:**
  + **Procedure Calls (ProcName(args))**:
    - Verifies that ProcName is declared as a PROCEDURE.
    - Checks that the number of actual arguments matches the number of formal parameters.
    - Checks that the type of each actual argument is compatible with the corresponding formal parameter type (including INTEGER to REAL compatibility).
  + **Function Calls (FuncName(args))**:
    - Verifies that FuncName is declared as a FUNCTION.
    - Checks argument number and type compatibility as with procedures.
    - The type of the function call expression is the declared return type of the function.
* **RETURN Statements (within Functions):**
  + Verifies that RETURN statements only appear within FUNCTION bodies.
  + The type of the expression in RETURN <expression>; must be compatible with the declared return type of the enclosing function (INTEGER can be returned for a REAL function type).
* **Built-in I/O Procedures:**
  + **write, writeln:**
    - Arguments can be expressions of type INTEGER, REAL, BOOLEAN, or string literals.
    - Errors are reported for unprintable argument types.
  + **read, readln:**
    - Arguments must be variables (l-values).
    - Arguments must be of a readable type (INTEGER or REAL).
    - Reading directly into an entire array identifier (e.g., read(myArray)) is an error; an element must be specified.

**3.3. Implementation via Visitor Pattern:**

* A SemanticAnalyzer class, implementing the SemanticVisitor interface, traverses the AST.
* Dedicated visit methods for each relevant AST node type implement the specific checks and AST annotations.

### **4. Key Successes in Semantic Analysis**

* **Robust Scope and Declaration Handling:** Successfully identifies undeclared variables and redeclarations, respecting global and local (subprogram) scopes.
* **Comprehensive Type Checking:** Accurate type determination for all expression types and enforcement of type compatibility rules across assignments, operations, and control structures.
* **Correct Subprogram Call Validation:** Ensures procedures and functions are called with the correct number and types of arguments. Validates return types for functions.
* **Array Semantics:** Correctly handles array declarations, type checking of array index expressions (must be INTEGER), and determines the type of an array element access. Errors for indexing non-array types.
* **AST Annotation:** Expression nodes are now correctly annotated with their resolved types, paving the way for subsequent phases like Intermediate Code Generation.
* **Specific Handling of Built-in I/O:** read, readln, write, writeln are recognized, and their unique argument requirements are semantically validated.
* **Clear Error Reporting:** Semantic errors are reported with descriptive messages, including line and column numbers, aiding in debugging MiniPascal source code.

### **5. Illustrative MiniPascal Code Examples (Semantically Validated)**

The compiler now correctly validates a wide range of MiniPascal programs, including those involving nested scopes, various data types, array manipulations, and subprogram calls. The examples from the previous report that were parsable are now also semantically checked. Additionally, complex constructs and error conditions are handled as demonstrated during testing.

**Example of a correctly analyzed complex program:**

PROGRAM ComplexSemanticTest;  
VAR  
 globalArr : ARRAY [0..9] OF INTEGER;  
 isValid : BOOLEAN;  
  
PROCEDURE Fill(limit : INTEGER);  
VAR i : INTEGER;  
BEGIN  
 i := 0;  
 WHILE i < limit DO  
 BEGIN  
 IF (i >= 0) AND (i < 10) THEN globalArr[i] := i \* i;  
 i := i + 1;  
 END;  
END;  
  
FUNCTION GetSum(maxIndex : INTEGER) : INTEGER;  
VAR tempSum, k : INTEGER;  
BEGIN  
 tempSum := 0;  
 k := 0;  
 WHILE k <= maxIndex DO  
 BEGIN  
 IF (k >= 0) AND (k < 10) THEN  
 tempSum := tempSum + globalArr[k];  
 k := k + 1;  
 END;  
 RETURN tempSum;  
END;  
  
BEGIN // Main  
 Fill(10);  
 isValid := GetSum(5) > 50;  
 IF isValid THEN  
 write('Sum was greater than 50.')  
 ELSE  
 write('Sum was not greater than 50, it was: ', GetSum(5));  
END.

*(This program would be successfully validated by the semantic analyzer).*

**Examples of code that now correctly generate semantic errors:**

* x := y; where y is undeclared.
* myInt := 3.14; where myInt is INTEGER.
* IF 10 THEN ... (condition not boolean).
* MyProc(1.0); if MyProc expects an INTEGER.
* FUNCTION F : INTEGER; BEGIN RETURN TRUE; END; (return type mismatch).

### **6. Current Language Limitations (Relevant to Semantics)**

While the semantic analyzer is robust for the current MiniPascal subset, the language itself (and thus the scope of semantic checks) has limitations that persist from the parsing phase:

* No CONST declarations.
* No user-defined TYPE declarations (e.g., enumerations, subranges, records). Semantic checks for these would be future work if the language expands.
* No CHAR data type.
* No advanced control flow: FOR loops, REPEAT...UNTIL loops, CASE statements.
* No explicit pass-by-reference for parameters (all are pass-by-value conceptually).
* Limited String Operations: Strings are literals; no string variables or complex operations. Built-in I/O handles string literals for output.
* Error Recovery: Focus is on error detection and reporting; sophisticated recovery to allow further analysis after an error is minimal.