TESTING REPORT – MINISquare25 Compiler

This report documents the testing carried out on my MiniSquare25 compiler implementation.

Testing covers all compiler stages:

Tokenising, Parsing, Declaration Identification, Type Checking, TAM Code Generation

Two full **MiniSquare** programs were used for testing:

- Example 1: **myprogram.tri**
- Example 2: **myprogram_example2.tri**

These programs exercise different language features and validate that the compiler works correctly.

1. Tokeniser Testing

The tokenizer was validated using both example files.

Results

The tokenizer correctly recognised:

Keywords (let, local, int, repeat, until, if, while, else, pass, var)

Identifiers

Integer literals

Character literals ("x" format required by MiniSquare25)

Operators (==, <, >, +, -)

Comments beginning with !

Brackets and braces

Whitespace and newlines, no invalid tokens produced during either example

The tokeniser correctly skipped comments and handled character literals according to the MiniSquare specification.

2. Parser Testing

Dispatcher traces confirmed that the Abstract Syntax Tree was built correctly for both examples.

Example 1 - myprogram.tri	**Example 2 - myprogram_example2.tri**
Parser produced correct nodes for:	Parser successfully handled:
Let declaration block	Multiple function calls (put, putint, getint, puteol)
Variable declaration	Character expressions
Assignment (x == x + 1)	Nested blocks
Repeat – until construct	Nested if statements
Relational expression (x > y)	While loops
	Var-parameter call (getint(var n))
	Arithmetic and relational expressions
	The grammar handled all constructs as expected with no syntax errors.

3. Declaration Identification Testing

Both examples confirmed correct scope and binding.

Correct Behaviour

All identifiers in both programs refer to the correct declarations.

Local variables declared inside let local are properly scoped.

var n in **Example 2** correctly resolves n to its declaration.

No undeclared-variable errors occurred.

This validates correct environment construction and identifier lookup.

4. Type Checking Testing

The type checker was exercised thoroughly by both programs.

Example 1 - myprogram.tri	**Example 2 - myprogram_example2.tri**
Arithmetic (x + 1) correctly typed as **int**	Character literals ("e", "n", etc.) typed as **char**
Relational comparison (x > y) typed as **bool**	Numeric expressions (n - 1) typed as **int**
The condition in until validated as **boolean**	Relational comparisons (n > 0, n < MAX) typed as **bool**
Assignment types (x == ...) matched correctly	getint(var n) successfully validated as a **var-**parameter
	All nested conditions and assignments passed type checking

Type checking produced no errors for either program once the var-parameter and character literal rules were implemented.

5. TAM Code Generation Testing

The code generator was validated using the generated TAM output from both examples.

Example 1 - myprogram.tri	**Example 2 - myprogram_example2.tri**
Memory allocation for local variables	Correct address loading for var parameters (LOADA SB+…)
Correct implementation of the repeat–until loop	Corrrect address loading for var parameters (LAODA …)
Arithmetic addition via CALL ADD	Input handling with CALL GETINT
Boolean comparison via CALL GT	Nested if/else compiled into correct conditional jumps
Backwards jump for looping	Decrement logic generated correctly (CALL SUB)
Correct program termination with HALT	Correct while-loop structure using conditional branches
The program executed correctly in the TAM machine.	The program executed correctly in TAM, displaying the countdown from the user's input down to 1.

Error Example Summary

Two additional programs: **myprogram_error** and **myprogram_error_2** were included to demonstrate how the compiler handles syntactic errors during the parsing stage.

Result When Running Both Error Examples

When compiled, both programs produced the following outputs respectively:

```
Compiling...
Tokenising...Done
Parsing...ERROR: Declaration must start with an identifier
ERROR: Invalid Command
Compilation failed with 2 errors
```

Explanation

These examples intentionally contain incorrect **MiniSquare** syntax. The parser correctly detects that:

- A declaration does not start with a valid identifier
- This violates the **MiniSquare** grammar rule requiring declarations to begin with a type followed by an identifier.
- An invalid command appears in the command sequence
- Because the earlier syntax error breaks the parser's expected structure, additional invalid-command errors are triggered.

Why These Errors Matter

These error examples confirm that:

- The parser correctly rejects malformed syntax
- Error reporting works properly
- The compiler stops safely when invalid constructs are encountered
- No further stages (identification, type checking, code generation) are run after a fatal parse error
- This completes coverage of negative testing, demonstrating that the compiler not only handles valid programs correctly but also fails safely with helpful diagnostics.