Compiler Construction and Type Inference

Matthew Ibrahim

Master of Science in Computer Science Capstone Testing

Grand Canyon University

Professor Aiman Darwiche

February 24, 2023

# Preface

The testing of MattyLang is driven primarily by code-coverage. The compiler is checked for correctness in two cases: when the source provided is valid, to ensure the result of compilation is expected with no diagnostics; and when the source provided is invalid, to test faultless parsing and the emitted diagnostics. At the time of this writing, MattyLang’s tests cover roughly 96% of the codebase. Ensuring the correctness of the compiler when the source code is invalid is vital to catch potential bugs that may not exist with a correct program. For example, ensuring the type-checking of an expression produces a diagnostic if the expression is not sound is important in ensuring type safety.

# Module Test Cases

MattyLang decouples various parts of the compiler into separate, easily reusable and testable modules, and comprises of: the Diagnostics class, the LineMap class, and the SymbolTable class. The tests for these modules can be found in *tests/test\_diagnostics.py*, *tests/test\_linemap.py*, and *tests/test\_symbols.py*, respectively. All tests are currently passing.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Diagnostics | tests/test\_diagnostics.py | High Priority |
| Test Objective | Ensure diagnostics of different types (info, warning, errors) can be added, and each set of diagnostics for phases of the compiler are sorted by position. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | has\_error() should return false if no error diagnostic has been emitted | has\_error() returns false if no error diagnostic has been emitted | Pass |
| 2 | has\_error() should return true if an error diagnostic has been emitted | has\_error() returns true if an error diagnostic has been emitted | Pass |
| 3 | next\_set() should split the diagnostics and sort the last set | After emitting info@1, warning@2, error@0, the diagnostics is reordered to error@0, info@1, warning@2 after next\_set is called | Pass |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | LineMap | tests/test\_linemap.py | High Priority |
| Test Objective | Ensure the mapping provided between source position and (line, column) are correct. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | get\_location(p) should return the line, column of position p | Given source "\nabc\ndef", running get\_location for each position correctly matches their expected location | Pass |
| 2 | get\_position(line, column) should return the position, given the line and column. | Given the locations generated in the previous test, they should remap back into their expected positions | Pass |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Symbols | tests/test\_symbols.py | High Priority |
| Test Objective | Ensure the symbol table handles block scoping and boundaries properly | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | lookup(name) should return None if no symbol exists with that name | Given an empty symbol table, lookup('a') returns None | Pass |
| 2 | register(name) should register a symbol with the given name and return the new symbol | Given an empty symbol table, register('a') returns a new symbol | Pass |
| 3 | lookup(name) should return the symbol if one exists with the name | Given the previous symbol table, lookup('a') returns its symbol | Pass |
| 4 | enclosing\_boundary() should return None if there is no ancestor symbol table that is a boundary | Given the previous symbol table, enclosing\_boundary() returns None | Pass |
| 5 | open\_scope(boundary) should open a new scope, and marks it as a boundary symbol table if True is passed, and returns the new symbol table | Given the previous symbol table, open\_scope(boundary=True) returns a new symbol table marked as a boundary | Pass |
| 6 | lookup(name, recursive, ignore\_boundary) should return None if the symbol is not part of the symbol table | Given the previous symbol table, lookup('a', False) returns None; lookup('a', True) returns None; lookup('a', True, ignore\_boundary=True) returns the symbol | Pass |
| 7 | Symbol.rename(name) should rename the symbol and update the associated symbol table | Given the previous symbol table, symbol\_a.rename('b') renames the symbol and updates it in the symbol table | Pass |
| 8 | Symbol.erase(name) should erase the symbol from the symbol table | Given the previous symbol table, symbol\_a.erase() erases the symbol from the symbol table | Pass |

# Requirements Testing

MattyLang requirements are tested by phase, in which each phase of compilation provides a specific requirement. This comprises of testing the Lexer class, Parser class, Binder visitor, Checker visitor, and Emitter visitor. The tests for these classes can be found in *tests/test\_lexer.py, tests/test\_parser.py, tests/test\_binder.py, tests/test\_checker.py, tests/test\_emitter.py*, respectively. All tests are currently passing.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Lexer | tests/test\_lexer.py | High Priority |
| Test Objective | Ensures the lexer can properly tokenize source code, and when it cannot it properly emits diagnostics and recovers. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | Scanning a source comprised of all the keywords and punctuations should return tokens representing them. | Keywords and punctuation are properly scanned, and the returned tokens meet what is expected and there are no diagnostics | Pass |
| 2 | Scanning a source comprised of identifiers, real literals, and string literals should return tokens representing them. | Identifiers, real literals, and string literals are properly scanned, and the returned tokens meet what is expected and there are no diagnostics | Pass |
| 3 | Scanning invalid lexemes should emit diagnostics for: missing digits around a decimal point (error), missing whitespace after a real literal (warning), unterminated string (error), and unexpected character (error) | Scanning the source: '.\n1.a\n# asd\n"asd\n;' properly emits these 4 diagnostics. | Pass |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Parser | tests/test\_parser.py | High Priority |
| Test Objective | Ensures the parser can properly parse source code, and when it cannot it properly emits diagnostics and recovers. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | Chunks, variable definitions, variable assignments should parse into their respective AST nodes | The source correctly parses into these AST nodes, and there are no diagnostics. | Pass |
| 2 | While, If, Else, Break, and Continue statements properly parse into their respective AST nodes | The source correctly parses into these AST nodes, and there are no diagnostics | Pass |
| 3 | Function definitions, function calls, and return statements properly parse into their respective AST nodes | The source correctly parses into these AST nodes, and there are no diagnostics | Pass |
| 4 | Primitive types and function types should properly parse into their respective AST nodes | The source correctly parses into these AST nodes, and there are no diagnostics | Pass |
| 5 | Syntactical errors should allow the compiler to recover and emit diagnostics to represent them | Syntax errors in the source do not prevent the parser from continuing, and diagnostics are produced and match what is expected | Pass |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Binder | tests/test\_binder.py | High Priority |
| Test Objective | Ensures the binder properly generates the symbol table and binds symbols to identifiers. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | Binding a semantically valid AST should produce no diagnostics, generate a symbol table, and bind symbols to identifiers | The symbol table generated passes what was expected, ensuring identifiers are bound to the correct symbols even in cases of a shadowed variable name within a sub chunk | Pass |
| 2 | Binding a semantically invalid AST should produce diagnostics, ensuring no variable duplications and undefined references | Given an AST comprised of duplicate definitions, duplicate parameters, and undefined references, diagnostics are produced and match what is expected | Pass |

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Checker | tests/test\_checker.py | Medium Priority |
| Test Objective | Ensures the checker properly performs type checking and inference and binds types to expressions. | | |
| Test Number | **Test Detail** | **Result** | **Status** |
| 1 | Checking a semantically valid AST should produce no diagnostics and bind types to expressions. | Type-checking is properly performed, with no diagnostics | Pass |
| 2 | Checking a semantically invalid AST should produce diagnostics | Given an AST comprised of type errors, such as assignment of a different typed value to a variable, non-bool typed conditions, inappropriate break/continue/return statements, and so on, produces diagnostics and match what is expected | Pass |

# System Testing

The MattyLang compiler front-end abstracts the process of compilation and provides options to control the process if needed. The front-end is implemented in *mattylang/\_\_init\_\_.py* and the primary interface is the compile function. In this case, the correctness of all phase of the compiler ensures correctness of the compiler, as each phase is dependent on a prior phase (excluding lexical analysis).

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Compiler and REPL | Medium Priority | |
| Test Objective | Ensures the compiler properly invokes all phases of the compiler as specified. | | |
| Test Number | **Test** | **Result** | **Status** |
| 1 | compile('examples/v0.3.py') properly compiles the file and returns an object containing the module, AST, and generated Python code | The CompileResult returned by compile contains all expected results, with the AST matching the expected AST of the source file, and the generated Python code was valid. | Pass |
| 2 | REPL-mode of the MattyLang front-end (matty.py) should allow for easy testing of features. | REPL mode works as expected, respecting all switches passed to the front-end. | Pass |