Compiler Construction and Type Inference

Matthew Ibrahim

Master of Science in Computer Science Capstone Implementation

Grand Canyon University

Professor Aiman Darwiche

February 1, 2023

# Table of Contents

[Table of Contents 2](#_Toc126894132)

[System Entities 3](#_Toc126894133)

[Top Level Classes 3](#_Toc126894134)

[Abstract Syntax Tree Topology 4](#_Toc126894135)

[Function Requirements 5](#_Toc126894136)

[Source Code Listing 5](#_Toc126894137)

[Code Review 6](#_Toc126894138)

[Implementation Plan 6](#_Toc126894139)

[Technologies 7](#_Toc126894140)

[Deployment 7](#_Toc126894141)

[Project Requirements Review 9](#_Toc126894142)

[Appendix A – Usage 10](#_Toc126894143)

[Appendix B – Grammar 11](#_Toc126894144)

# System Entities

## Top Level Classes

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Module | | + file: str | | + source: str | | + diagnostics: Diagnostics | | + globals: SymbolTable | | |  | | --- | | Lexer | | + scan() -> Token |  |  | | --- | | Parser | | + parse() -> ProgramNode | | | | |  | | --- | | Token | | + kind: str | | + lexeme: str | | + position: int | |
| |  | | --- | | AbstractNode | | + parent: Optional[AbstractNode] | | + position: int | | + invalid: bool | | + accept(visitor: AbstractVisitor) -> None | | + get\_first\_ancestor(predicate: Callable[[AbstractNode], bool]) -> Optional[AbstractNode] |  |  | | --- | | LineMap | | + get\_position(line: int, column = 1) -> str | | + get\_line(position: int) -> int | | + get\_column(position: int) -> int | | + get\_location(position: int) -> Tuple[int, int] | | | |  | | --- | | AbstractVisitor | | + visit\_program(node: ProgramNode) -> None | | + visit\_chunk(node: ChunkNode) -> None | | + ... (see Figure 1) |  |  | | --- | | ScopedVisitor : AbstractVisitor | | + visit\_program(node: ProgramNode) -> None | | + visit\_chunk(node: ChunkNode) -> None | | + ... (see Figure 1) |  |  | | --- | | Diagnostics | | + has\_error() -> bool | | + emit\_diagnostic(kind: str, message: str, position = 0) -> None | | + sort -> None | | | |
| |  | | --- | | SymbolTable | | + parent: Optional[SymbolTable] | | + children: List[SymbolTable] | | + symbols: Dict[str, Symbol] | | + boundary: bool | | + lookup(name: str, recursive = True, ignore\_boundary = False) -> Optional[Symbol] | | + register(declaration: Declaration) -> Symbol | | + open\_scope(boundary: bool = False) -> SymbolTable | | + close\_scope() -> Optional[SymbolTable] | | | | |  | | --- | | Symbol | | + kind: str | | + lexeme: str | | + position: int |  |  | | --- | | Declaration | | + get\_declared\_name() -> str | | + lexeme: str | | + position: int | | |

## Abstract Syntax Tree Topology

Background pattern

Description automatically generated with low confidence

Figure 1: The inheritance tree of the MattyLang AST nodes. Variable definition and function definition nodes also inherit from the Declaration top-level class.

# Function Requirements

|  |  |  |
| --- | --- | --- |
| ID | Name | Features |
| 1 | Efficient data structure mapping positions to its corresponding line/column tuple. | LineMap |
| 2 | Diagnostics to keep track of compiler info/warnings/errors. | Diagnostics |
| 3 | Lexical analysis phase of the compiler. | Lexer, Tokens |
| 4 | Syntax analysis phase of the compiler. | Parser, Nodes |
| 5 | Semantic analysis phase of the compiler. | Binder, Checker Visitors Symbol Table |
| 6 | Code generation phase of the compiler. | Emitter Visitor |
| 7 | Compiler front-end: command-line interface with REPL. | Entry Point |
| 8 | v0.1 features: chunks, block-scoped variables, basic expressions and statements. | \* |
| 9 | v0.2 features: control flow statements | \* |
| 10 | v0.3 features: functions and function types | \* |

# Source Code Listing

|  |  |
| --- | --- |
| Name | Description |
| docs/grammar.ebnf | EBNF-like definition of the MattyLang syntax |
| docs/ProjectProposal.docx  docs/ProjectArchitecture.docx docs/Implementation.docx  docs/UserGuide.docx  docs/TestCases.xlsx | Project Documentation and Rationale |
| examples/\*.mtl | Example MattyLang source codes |
| mattylang/ | Compiler implementation |
| mattylang/ast/\*.py | Abstract syntax tree node definitions |
| mattylang/visitors/printer.py | Printer visitors, used for --syntax, --symbols CLI options |
| mattylang/visitors/binder.py | Generates the symbol table and binds symbols to names, also handles lexical scoping |
| mattylang/visitors/checker.py | Performs type checking and type inference, and binds types to applicable nodes |
| mattylang/visitors/emitter.py | Performs code generation, specifically transpilation to Python3 |
| mattylang/diagnostics.py | Classes used to store compiler diagnostics (such as warnings and errors) |
| mattylang/lexer.py | Provides a token stream to be used by the parser; lexical analysis |
| mattylang/linemap.py | Provides an efficient data structure to map source code position to their respective line and column |
| mattylang/module.py | Top-level abstraction of a module, which is comprised of a file name and source code |
| mattylang/nodes.py | Node class amalgamation (for easy imports) |
| mattylang/parser.py | Transforms a token stream into an undecorated abstract syntax tree; syntax analysis |
| mattylang/symbols.py | Provides the implementation of the Symbol Table, tracks declarations and handles lexical scoping |
| mattylang/visitor.py | Base class for visitors, whose methods visit all chilren of a node |

# Code Review

|  |  |
| --- | --- |
| Date | October 21, 2022 |
| Files | mattylang/ast/\*.py |
| Summary | Replace line, column fields with a position field to reduce memory usage of nodes. |
| Change | Replaced line and column node fields with position, introduce LineMap to provide an efficient mapping between a 0-indexed position in the source code. The LineMap data structure only uses memory proportional to the number of newline characters in a file, and a position to line, column pair uses time proportional to the log (base 2) of the number of newline characters. This reduces total memory usage (and reduces parser complexity) by an order of one integer per node class. |

|  |  |
| --- | --- |
| Date | January 3, 2023 |
| Files | mattylang/symbols.py, mattylang/ast/core.py, mattylang/ast/statement.py |
| Summary | Generalize symbols to refer to an abstract Declaration class that declaration/definition nodes inherit from to decouple concrete declaration nodes from the symbol table. |
| Change | Create abstract Declaration class, have variable definition and function definition nodes inherit from it, and make the Symbol class refer to a Declaration instead of a VariableDefinitionNode or FunctionDefinitionNode. This decouples the concrete definition node classes from the symbol table. |

|  |  |
| --- | --- |
| Date | February 5, 2023 |
| Files | doc/grammar.ebnf, mattylang/ast/\*.py |
| Summary | Introduce VariableDeclaration node type to use for function parameters. |
| Change | Create VariableDeclaration class that inherits Declaration, contains a name (IdentifierNode) field and a type (TypeNode) field. These behave as explicitly typed, to-be-initialized variables. |

# Implementation Plan

The MattyLang parser is a handwritten, recursive-descent compiler that utilizes top-down operator precedence parsing to handle most expression parsing. The visitor pattern is used to provide a traditional interface to provide abstract syntax tree (AST) traversal. The root class of all visitors, AbstractVisitor, provides default visitors for all node types that merely visit its children. Semantic checking and code generation will provided through the use of visitors. MattyLang v1.0 is the project goal, and a runtime/interpreter will not be provided as MattyLang source code will compile into Python3 code. A custom runtime will likely be provided in the future. However, translating to Python3 presents a couple of unique challenges. For instance, Python3 does not support lexical scoping, but MattyLang does. Therefore, variables must be renamed if there is a variable with the same name in a parent scope (up to the first enclosed function definition).

## Technologies

|  |  |
| --- | --- |
| Technology | Description |
| Python3 | Host language |
| Python3 Standard Library | argparge (providing CLI argument parsing), typing, bisect (for efficient upper-bound binary search when querying a LineMap), abc (for abstract classes), string, and unittest. |
| TextMate | To provide basic syntax highlighting for MattyLang, in the form of a tmLanguage. |
| Visual Studio Code and Extensions | Code editor, with extensions to provide Python linting, formatting, editorconfig support. |
| Git/GitHub | Version control system, with repository publicly hosted on GitHub. GitHub Actions are also used for continuous integration, to provide testing and upload code coverage. |
| Codecov | To determine how much of the codebase is covered given the tests. |

## Deployment

The MattyLang compiler is a Python3 library with a front-end to provide both a read-eval-print -loop (REPL) as well as file compilation and options to help visualize the outputs of the different compiler components. The top-level *matty.py* file is the script that provides the front-end, and the files in the *mattylang* folder provide the library implementation that can be used directly if needed to implement linters and formatters, refactoring tools, alternative front ends, and so on. For a quick usage guide, the front-end (*matty.py*) can be invoked with the --help option (see Appendix A – Usage). More information on the usage of the MattyLang compiler can be found in the User Guide document.

The compiler is separated into distinct phases (see Figure 2: Compiler Phases): lexical analysis (provided by the Lexer class), syntax analysis (provided by the Parser class), semantic analysis (two stage: symbol binding, provided by the Binder visitor; and type checking, provided by the Checker visitor), and code generation (provided by the Emitter visitor). The symbol binding stage of semantic analysis generates a module’s symbol table and handles lexical scoping rules of lookups. The type checking stage of semantic analysis depends on the symbol binding stage and provides type checking and binds types to applicable untyped nodes (i.e., type inference). In a valid MattyLang program, all identifiers are bounded to a declaration through a symbol (a symbol encapsulates the name, type, and location of a declaration).

Diagram

Description automatically generated

Figure 2: Compiler Phases

# Project Requirements Review

MattyLang has undergone a couple of transformations since its conception:

1. Initially, MattyLang’s host language was Lua. Due to the lack of typing support, and the fact that Python is more popular, the host language was switched to Python3.
2. An interpreter was initially part of the MattyLang proposal, with no code generation phase planned. Execution of MattyLang source code was to be interpreted by traversing the AST to compute expressions, follow function calls, handle other control flow statements, and so on. However, the scope of MattyLang v1.0 is on compiler construction and type systems, so a code generation stage was introduced with the target output being Python3 and the interpreter was scrapped. This scaled down the project quite significantly due to unforeseen complexities when designing the interpreter.
3. The Lexer and Parser classes were originally planned to be decoupled, then planned to be combined as the output of the Lexer is rarely ever useful, but this plan was discarded and the Lexer and Parser currently remain decoupled.
4. Expressions are not statements, as to eliminate as much ambiguity in the grammar as possible and keep the parser as simple as possible. There is one oddity: function calls may appear where both a statement or expression is expected. To handle this case, a CallStatement was introduced which merely holds a reference to a CallExpression.

The MattyLang API provides a simple interface, allowing compilation and AST traversal regardless of the existence of the front-end. All values in MattyLang are of type: Nil, Bool, Real, String, or of a function type. Functions in MattyLang may be higher-order functions (i.e., functions that can accept and/or return functions) and mutual recursion is supported when the return type can be inferred.

# Appendix A – Usage

The front-end to the *MattyLang v1.0* is a command-line interface with the following usage:

usage: matty.py [-h] [-o OUTPUT] [-V] [-v] [--tokens] [--syntax] [--symbols] [--code] [--no-analysis] [file]

MattyLang frontend, compiles and executes MattyLang files.

positional arguments:

file the input file (none for REPL)

options:

-h, --help show this help message and exit

-o OUTPUT, --output OUTPUT

the output file

-V, --version show program's version number and exit

-v, --verbose verbose output

--tokens print the tokens

--syntax print the syntax tree

--symbols print the symbol table

--code print the generated code

--no-analysis skip semantic analysis (implies no codegen)

# Appendix B – Grammar

The syntactical grammar of *MattyLang v1.0* is defined, in an EBNF-like fashion, as follows:

program = chunk EOF; (\* v0.1 \*)

(\* abstract \*)

statement = "{" chunk "}" | variable\_definition | variable\_assignment; (\* v0.1 \*)

statement = if\_statement | while\_statement | break\_statement | continue\_statement; (\* v0.2 \*)

statement = function\_definition | return\_statement | call\_statement; (\* v0.3 \*)

(\* v0.1 \*)

chunk = { statement };

variable\_definition = "def" identifier "=" expression;

variable\_assignment = identifier "=" expression;

(\* v0.2 \*)

if\_statement = "if" "(" expression ")" statement {"elseif" "(" expression ")" statement} ["else" statement];

while\_statement = "while" "(" expression ")" statement;

break\_statement = "break";

continue\_statement = "continue";

(\* v0.3 \*)

function\_definition = "def" identifier "(" [parameter { "," parameter } [","]] ")" "{" chunk "}";

parameter = identifier ":" type;

return\_statement = "return" [expression];

call\_statement = call\_expression;

(\* abstract \*)

expression = "(" expression ")"; (\* v0.1 \*)

expression = primary\_expression | unary\_expression | binary\_expression; (\* v0.1; v0.3 \*)

(\* v0.1 \*)

unary\_expression = ("-" | "!") expression;

binary\_expression = expression ("+" | "-" | "\*" | "/" | "%" | "<" | ">" | "<=" | ">=" | "==" | "!=" | "||" | "&&") expression;

(\* abstract \*)

primary\_expression = nil\_literal | bool\_literal | real\_literal | string\_literal | identifier; (\* v0.1 \*)

primary\_expression = call\_expression; (\* v0.3 \*)

(\* v0.1 \*)

nil\_literal = "nil";

bool\_literal = "true" | "false";

real\_literal = DIGIT { DIGIT } "." { DIGIT } | "." DIGIT { DIGIT };

string\_literal = "'" { GRAPHICAL | " " | "\t" } "'" | '"' { GRAPHICAL | " " | "\t" } '"';

identifier = { ALPHABETICAL | "$" | "\_" } { ALPHANUMERIC | "$" | "\_" };

(\* v0.3 \*)

call\_expression = identifier "(" [expression { "," expression } [","]] ")";

(\* abstract \*)

type = primitive\_type; (\* v0.1 \*)

type = function\_type; (\* v0.3 \*)

(\* v0.1 \*)

primitive\_type = "Nil" | "Bool" | "Real" | "String";

(\* v0.3 \*)

function\_type = "(" [{ type "," } type [","]] ")" "->" type;

Quoted text represents terminals (tokens), and the grammar production rules are nearly one-to-one with the AST nodes. Text surrounded in (\* ... \*) represent comments, typically indicating if a node is abstract or what version the node is implemented in.