Compiler Construction and Type Inference

Matthew Ibrahim

Master of Science in Computer Science Capstone Project Architecture

Grand Canyon University

Professor Aiman Darwiche

January 25, 2023

# **Abstract**

The evolution of software has given developers an arsenal of tools, libraries, and abstractions that allows for increased productivity and reduces the barrier of entry for new programmers to begin to develop software. Much of this has been made possible by the decoupling of hardware and software. Software engineers can find success in software development without the need for a deep understanding of hardware. Also, software can be built on and deployed to many different platforms and architectures. To allow for such decoupling, developers must be able to write hardware-agnostic code, and various abstractions must be in place to make this possible. A specific area of improvements relates to the creation of modern programming languages, such as Go, Rust, and Python; and continuous improvements in existing ones, such as Java and C++. The increasing complexity of software and programming languages necessitates increasingly complex compilers (the software responsible for transforming source code into an intermediate, portable, or executable form) to aid developers and ensure correctness. One vital tool to aid in software development has been the evolution of type systems, particularly: the ability to provide the compiler with more semantic information based on the values a variable can hold. This allows a set of bugs to be caught before execution, at compile-time. Type systems save time, money, and effort. In this project, the focus is on the design and construction of a compiler with nominal types, and support for type inference. Specifically, a compiler for a custom, statically typed programming language called *MattyLang* will be the product of this project.

# Table of Contents

[Abstract 2](#_Toc125586135)

[Table of Contents 3](#_Toc125586136)

[Design Planning Summary 4](#_Toc125586137)

[Overview of Design Concepts 5](#_Toc125586138)

[Detailed Solution Architecture 7](#_Toc125586139)

[User Interface 7](#_Toc125586140)

[Compiler 7](#_Toc125586141)

[Class Diagrams 9](#_Toc125586142)

[AST Node Hierarchy 10](#_Toc125586143)

[Appendix A – Grammar 11](#_Toc125586144)

# Design Planning Summary

The MattyLang programming language is a nominal, statically typed custom programming language whose host will be the Python programming language. In short, the product can be broken down into a tree of tasks, comprising the Design phase, the Develop phase, and the Analysis phase (see Figure 1). Given the increasing utility and reliance on type systems in modern programming languages, their design and implementation has become just as vital, if not more, than many other component of a compiler. Understanding how type systems are designed and implemented will allow developers to take better advantage of typed programming languages and adapt to other languages more quickly.

This project aims to explore the design and implementation of both compilers and type systems by producing a beginner-friendly, traditional, and concise implementation of a compiler with support for type inference. In future versions (beyond the scope of this project), many additional features will be developed¸ such as compound types, tuples, closures, and so on, though they are beyond the scope of *MattyLang v1.0*, the final product of this project.

A picture containing diagram

Description automatically generated

Figure : Project Overview

# Overview of Design Concepts

This project will not utilize the typical strategy of implementing one part of the compiler at a time. Rather, a minimal subset of the language will be fully implemented, and features will be incrementally added until all the specifications are met. To ensure correctness, end-to-end tests will be used to ensure both the correctness of the compiler for both valid, and invalid inputs. This methodology allows for quicker prototyping and helps to quickly narrow the interface of the various components of the compiler sooner rather than later. The initial language is versioned as 0.1.0, with each subsequent additional feature incrementing the minor version. Once the final version is reached, the major version will be incremented. The initial version (v0.1) implements variables, expressions, primitive literals, and primitive types. The proceeding version (v0.2) is a superset of v0.1, and includes support for many control flow statements. The third version (v0.3) is a superset of v0.2, and includes support for functions and function types.

The front-end is a CLI, whose usage is as follows: (generate by invoking $ *matty.py -h*)

Snippet 1: Command-line interface, generated with Python3’s argparse.

usage: matty.py [-h] [-o OUTPUT] [-V] [-v] [--tokens] [--syntax] [--symbols]

[--code]

[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

The compilation phase works by first parsing a *MattyLang* source file (in which a Lexer is invoked to continuously reduce the source code into a stream of tokens), which produces an *undecorated* abstract syntax tree. Following that, the *binder* visitor binds symbols (which keep track of declarations and references) to identifiers. Following that, the *checker* visitor binds types to expressions and symbols. Finally, the *emitter* visitor will emit *Python3* code that is semantically equivalent to the original *MattyLang* source. In future versions of *MattyLang*, a runtime will be designed and implemented, and other features will be implemented to cover more aspects of programming language design. The following flowchart encapsulates the logical model of the compiler phase:

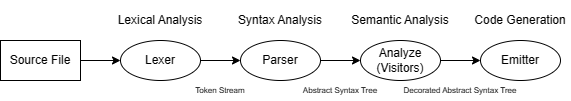


Figure 2: Compiler Model

# Detailed Solution Architecture

The development phase of the project is comprised of the implementation of the compiler and the user interface. The *Python3* programming language will be used to implement this project, utilizing only the Python3 standard library. For development, continuous integration through GitHub Actions will be used to provide testing and code coverage after every commit. Code coverage will be handled by *Codecov*, tests through the built-in Python package *unittest*, and a *TextMate* language will provide basic syntax highlighting for *MattyLang* (*\*.mtl*) source files.

## User Interface

The user interface (i.e., front end) is a CLI that invokes the *MattyLang* compiler compile files, and provide other features. The CLI must minimally support the following:

* REPL mode if no INPUT file specified.
* The INPUT file is the first positional command-line argument.
* The OUTPUT file is an option (*-o/--output)* used to specify the file to write the output of the compiler, which is *Python3* code.
* The SYNTAX switch (*--syntax*) will print the abstract syntax tree.
* The HELP switch (*-h/--help*) will print the usage.

Python3’s *argparse* library will be used to parse command-line arguments and generate the usage message.

## Compiler

The compiler (i.e., the back end) is a library that provides an interface to compile and manage *MattyLang* programs. The compiler will be implemented without the use of generator tools (e.g., GNU’s *lex* and *yacc*), by using recursive descent for general parsing and top-down operator precedence parsing to parse unary and binary expressions. The choice of utilizing a different parser strategy when parsing expressions is to combat the potential of deep grammatical nesting that may occur if the operator precedence was specified through the grammar. Additionally, the parser procedures and data types should match up nearly one-to-one with the grammar of the language (see *Appendix A - Grammar*). Furthermore, the compiler should be *faultless*. If there are lexical, syntax, or semantic errors, compilation should recover in some meaningful way and continue in order to provide as many diagnostics to the user as possible.

The **Lexer** class is responsible for providing lexical analysis. *MattyLang* code is transformed into a token stream, which allows tokenization to be decoupled from parsing.

The **Parser** class is responsible for providing syntax analysis. A token stream is transformed into a tree structure, describing the source code in a hierarchal, tree structure that can then be analyzed. The output of the parser is an undecorated AST and merely represents the syntactical structure of a program.

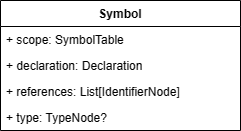
The **Binder** visitor is responsible for binding symbols to identifiers, in order to record references to variables. Aside from assigning to the symbol field of applicable AST nodes, a symbol table is generated to represent a hierarchal, tree structure of variable definitions. This table handles scoping rules and provides boundaries at every function definition, to ensure variables referenced external to the enclosed function can be properly handled. In v1.0, there is no support for closures and thus no support for referencing external variables, an undefined variable diagnostic will be produced instead. A symbol represents a declaration, and contains the declaration node, the declaration type, and references to all nodes that reference the declaration.

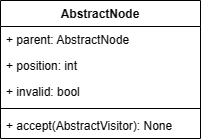
The **Checker** visitor is responsible for type-checking (ensuring type compatibility from the top-down) and type-inference (assigning types to expression nodes from the bottom-up). The checker must be invoked after the binder, and the output of the checker is a fully decorated AST. This tree can then be used for code generation, optimization, transformation, and so on.

The **Emitter** visitor is responsible for code-generation. In *MattyLang v1.0*, it emits *Python3* code that retains the semantic meaning of the original *MattyLang* source code. The emitter should be the last visitor invoked, as it is the final output of the compiler.

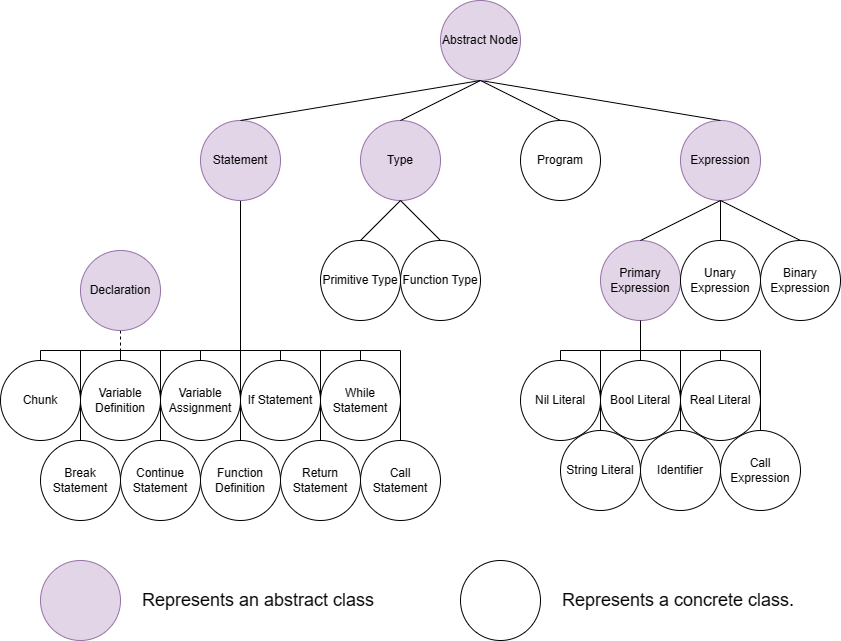
## Text Description automatically generatedText Description automatically generatedClass Diagrams



Text

Description automatically generated

## AST Node Hierarchy



## Technologies

1. Python 3 and development packages: unittest, coverage (for CI)
2. GitHub Actions with packages for CI: checkout@v3, setup-python@v4, weibullguy/python-lint-plus@master, codecov/codecov-action@v3
3. Visual Studio Code (Microsoft), and extensions, for development:
   1. EBNF Tools Extension (Ivaylo Gochkov)
   2. EditorConfig for VS Code (EditorConfig)
   3. Python/Pylance (Microsoft)
4. TextMate, grammar file to provide syntax highlighting for MattyLang source files.

# Appendix A – 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 "(" [identifier ":" type { "," identifier ":" type } [","]] ")" block;

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;