**MattyLang Compiler**

**Version 1.0**

**User Guide**

**MattyLang Compiler**

**User Guide**

Copyright © 2023 Matthew Ibrahim

All rights reserved.

# Preface

The MattyLang User Guide explains the usage of the MattyLang compiler and the command-line interface (CLI), that can be used to compile MattyLang source code; as well as the usage of the MattyLang compiler application programming interface (API), which can be used to interact with the compiler directly from Python. The primary purpose of the design and implementation of MattyLang is to explore compiler construction and type systems, with a focus on implementing a traditional compiler by hand. This compiler should be usable as a learning tool and should also be easy to expand upon. Furthermore, the use of parser generating tools such as *yacc*, *bison*, and so on are not used, as the purpose of MattyLang is to implement all phases of compilation, from lexical and syntax analysis to type checking and code generation. Throughout the rest of this guide, MattyLang specifically refers to MattyLang v1.0, unless otherwise stated.

# Table of Contents

[Preface 2](#_Toc127932074)

[Introduction 2](#_Toc127932075)

[Table of Contents 3](#_Toc127932076)

[General Information 4](#_Toc127932077)

[MattyLang CLI 5](#_Toc127932078)

[Getting Started 5](#_Toc127932079)

[Usage 6](#_Toc127932080)

[MattyLang Features 7](#_Toc127932081)

[Expressions 7](#_Toc127932082)

[Variables 7](#_Toc127932083)

[Control Flow 8](#_Toc127932084)

[Functions 8](#_Toc127932085)

[MattyLang API 8](#_Toc127932086)

[Front-End 8](#_Toc127932087)

[Visitors 9](#_Toc127932088)

[Frequently Asked Questions 10](#_Toc127932089)

[Help and Contact Details 11](#_Toc127932090)

[Appendix A –Usage 12](#_Toc127932091)

[Appendix B – Grammar 13](#_Toc127932092)

[Appendix C – Operator Precedence Table 15](#_Toc127932093)

[Appendix D – Source Code Listing 16](#_Toc127932094)

# General Information

MattyLang is a procedural, statically typed programming language with support for block scoped variables, functions and higher-order functions, branching/control flow statements, and arithmetic, logical, and relational expressions. The MattyLang compiler is written in Python 3 and makes heavy use of Python type annotations to help ensure correctness. The compiler also provides faultless parsing, which means the compiler will attempt to continuously parse the source code even after there are errors in the source code. Since the parser does not quit at the first error, the compiler is able to catch multiple compile errors instead of just the first one.

This guide can be split into the following sections:

1. The usage of the MattyLang command-line interface (CLI), which provides an interface to the compiler, and examples of various commands.
2. A summary of the features of MattyLang, with examples.
3. A summary of invoking the MattyLang Application Programming Interface (API), the compiler back-end, with examples to invoke the parser and create new visitors.
4. Additional documentation, such as a frequently asked questions and a glossary.

# MattyLang CLI

## Getting Started

MattyLang can be downloaded from <https://github.com/cntkillme/matty-lang>, and requires Python 3. It is recommended to change the working directory of your terminal to the root directory of the project, as that is what the rest of this guide will assume. It is the same directory comprising of the file matty.py, which is the CLI. To get started, invoke the command $ python matty.py examples/v0.3.mtl. The generated python file will be created in the working directory and can be executed with Python3.

|  |  |
| --- | --- |
| **file:** ./examples/v0.3.mtl | **file:** ./v0.3.py (CREATED) |
| def PI = 3.141592654  def fib(n: Real) {  if (n > 1)  return fib(n - 1) + fib(n - 2)  return 1  }  print("PI")  print(PI)  print("fib(5)")  print(fib(5)) # 8 | PI = 3.141592654  def fib(n):  if n > 1.0:  return fib(n – 1.0) + fib(n – 2.0)  return 1.0    print("PI")  print(PI)  print("fib(5)")  print(fib(5)) |
| **MattyLang $** python matty.py examples/v0.3.mtl  PI  3.141592654  fib(5)  8 | |

Since the compilation was successful, no diagnostics were produced. We can also enter a very simple read-eval-print-loop (REPL) mode by not specifying any file to the CLI:

**MattyLang $** python matty.py

> print("Hello, world!")  
Hello, world!

> def x = 1 + "a

stdin:1:15: error: syntax: expected " to terminate string

stdin:1:9: error: analysis: incompatible operand types for expression: Real + String

> def x = y

stdin:1:9: error: analysis: undefined variable y

Unlike other REPLs, the entire source must be entered at once. The EOF signal can be sent (Linux/Unix: Ctrl + D, Windows: Ctrl + Z + Return) to exit.

## Usage

The usage of the CLI can be generated by invoking $ python matty.py -h. The entire usage can be seen in Appendix A – CLI Usage. To analyze the output of specific compiler components, there are a variety of options that can be used in combination with each other, and can be used in REPL mode:

* The --tokens option will print the tokens making up the file.

|  |
| --- |
| **MattyLang $** python matty.py --tokens  > def x = 1  stdin:1:1: def  stdin:1:5: identifier(x)  stdin:1:7: =  stdin:1:9: real\_literal(1) |

* The --syntax option will print a readable version of the abstract syntax tree (AST).

|  |
| --- |
| **MattyLang $** python matty.py --syntax  > def x = 1  program <1, 1>  chunk <1, 1>  variable\_definition <1, 1>  identifier(x) <1, 5>  - symbol: symbol(name: x, node: variable\_definition, type: Real, extern: False)  - type: Real  real\_literal(1.0) <1, 9>  - type: Real |

* The --symbols option will print a readable version of the symbol table.

|  |
| --- |
| **MattyLang $** python matty.py --symbols  > def x = 1 def y = 2 { def x = "a" y = y + 1 } symbol(name: print, node: None, type: (Any) -> Nil, extern: True)  symbol(name: x, node: variable\_definition, type: Real, extern: False) <1, 1>  symbol(name: y, node: variable\_definition, type: Real, extern: False) <1, 11>  symbol(name: x\_1, node: variable\_definition, type: String, extern: False) <1, 23> |

An *extern* symbol refers to a symbol whose definition is not within the program, and the top-level symbol table represents the globals (only print for now) of the program.

# MattyLang Features

## Expressions

MattyLang supports arithmetic, relational, logical binary expressions, as well as common primary and unary expressions. The operator precedence table can be seen in Appendix C – Operator Precedence Table. In contrast to many other languages, expressions and statements are mutually exclusive; an expression cannot appear where a statement is expected, and vice-versa. A function call, however, is both an expression and a statement (internally represents as two distinct AST nodes).

|  |
| --- |
| **MattyLang $** python matty.py  > print(1 + 2 \* 3) # evaluated as 1 + (2 \* 3)  7.0 |

Aside from number literals, there exists the nil literal, two bool literals (true and false) and string literals. Each expression in MattyLang is typed, and these expressions are type checked during the semantic analysis phase of the compiler (see: checker.py).

## Variables

MattyLang supports block scoped variables, which allows for variables with the same name to be defined within different blocks. This contrasts with Python, which has function scoping. A simple demonstration makes this apparent:

|  |  |
| --- | --- |
| **file:** src.mtl | **file:** src.py |
| def x = 0  {  def x = 1  print(x) # 1  }  print(x) # 0 | x = 0  if True:  x = 1  print(x) # 1  print(x) # 1 |

## Control Flow

MattyLang supports if statements, while loops, and the break and continue statement. These statements work functionally identically as they do in many other languages. However, as MattyLang is a strongly typed language, conditions must evaluate into a boolean.

|  |
| --- |
| **MattyLang $** python matty.py  > if (4 == 3) print("4 == 3") else if (4 == 4) print("4 == 4")  4 == 4  > def sum = 0 def n = 1 while (n <= 10) { sum = sum + n n = n + 1 } print(sum) 55.0 |

## Functions

MattyLang supports functions, including higher order functions, as well as recursion. However, MattyLang does not yet support closures: attempting to access a non-function variable declared outside of a function will produce an undefined reference error. The return type of functions are inferred by the compiler, though the parameters must be explicitly typed.

|  |
| --- |
| **file:** hof.mtl |
| def apply(f: (Real, Real) -> Real, x: Real, y: Real) {  return f(x, y)  }  def add(x: Real, y: Real) { return x + y }  print(apply(add, 5, 10)) # 15 |
| **MattyLang $** python matty.py hof.py  15.0 |

Due to the current scoping rules, a function may only be called after it is defined. This makes mutual recursion, that is two functions that call each other, impossible in MattyLang.

# MattyLang API

## Front-End

The MattyLang API front-end, not to be confused with the MattyLang front-end (the CLI tool), exposes helper functions that abstract MattyLang compiler details. The result of the compile function is an object that contains the abstract syntax tree and generated Python code. The signature of the compile function is:

def compile(file: str, source: str, verbose: bool = False, parse\_only: bool = False, globals: Optional[SymbolTable] = None) -> CompileResult

|  |
| --- |
| **file:** test.py |
| from mattylang import compile  from mattylang.visitors.printers import AstPrinter  result = compile("test", "print(0)")  result.ast.accept(AstPrinter(result.module)) |
| **MattyLang $** python test.py  program <1, 1>  chunk <1, 1>  call\_statement <1, 1>  call\_expression <1, 1>  - type: Nil  identifier(print) <1, 1>  - symbol: symbol(name: print, node: None, type: (Any) -> Nil, extern: True)  - type: (Any) -> Nil  real\_literal(0.0) <1, 7>  - type: Real |

If compile is called in parse-only mode, it can be checked later by passing the compile result to the check function.

## Visitors

Semantic analysis, which comprises of symbol binding/name resolution and type checking, are implemented using the visitor design pattern. Custom visitors can be implemented by inheriting from the abstract visitor class located in visitor.py. By default, all visitor methods will recursively invoke the visitor on all the children of the node. Here is an example of a visitor that prints the names of all function definitions:

|  |
| --- |
| **file:** test.py |
| from mattylang import compile  from mattylang.ast import \*  from mattylang.visitor import AbstractVisitor  class PrintFunctionNames(AbstractVisitor):  def \_\_init\_\_(self):  super().\_\_init\_\_()  def visit\_function\_definition(self, node: 'FunctionDefinitionNode'):  print(f'function: {node.identifier.value}, type: {node.identifier.get\_symbol().get\_type()}')  super().visit\_function\_definition(node) # visit children  result = compile("test", "def f(){} def g(x: Real){}")  result.ast.accept(PrintFunctionNames()) |
| **MattyLang $** python test.py  function: f, type: () -> Nil  function: g, type: (Real) -> Nil |

# Frequently Asked Questions

1. Why is there no custom MattyLang runtime if this project aims to explore compiler design?
   1. The initial plan was to execute MattyLang code by traversing the AST. However, the initial version of MattyLang seeks only to explore the compilation stage with a focus on type checking. As such, spending time designing and implementing a runtime was scraped due to control flow statements and functions making a simple traversal of the AST insufficient. Future versions of MattyLang may introduce a custom runtime.
2. Why does MattyLang not support closures, given the output language is Python which does support closures?
   1. Once Python was decided as the output of the compiler, closures were initially planned to be supported. However, the implementation of functions took more time than expected, and so the feature was postponed.
3. Why is there no distinct integer type?
   1. No integer type was introduced yet because its implementation adds no real value to the project and introduces various complexities that are not worth dealing with. For example, conversion between the integer type and real type will have to be supported, either implicitly when safe or explicitly. Furthermore, integers without bitwise operators make such a feature less useful anyways.

# Help and Contact Details

To submit bug reports or file feature requests, please use the issue tracker in the MattyLang GitHub repository. Contributions and improvements are also welcome. For additional information, questions, or feedback, feel free to contact me at [mibrahi7@my.gcu.edu](mailto:mibrahi7@my.gcu.edu).

# 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] [--parse-only] [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 (default is <file>.py)

-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

--parse-only skip semantic analysis and code generation

# Appendix B – Grammar

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

program = { statement } EOF;

(\*\*\* statements \*\*\*)

(\* abstract \*)

statement = "{" { statement } "}"; (\* compound statement \*)

statement = variable\_definition | variable\_assignment;

statement = if\_statement | while\_statement | break\_statement | continue\_statement;

statement = function\_definition | return\_statement | call\_statement;

(\* concrete \*)

chunk = statement | "{" { statement } "}";

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

variable\_assignment = identifier "=" expression;

if\_statement = "if" "(" expression ")" chunk ["else" chunk];

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

break\_statement = "break";

continue\_statement = "continue";

function\_definition = "def" identifier "(" [variable\_declaration { "," variable\_declaration } [","]] ")" "{" { statement } "}";

function\_parameter = identifier ":" type;

return\_statement = "return" [expression];

call\_statement = call\_expression;

(\*\*\* expressions \*\*\*)

(\* abstract \*)

expression = "(" expression ")";

expression = primary\_expression | unary\_expression | binary\_expression; (\* operator-precedence parsing \*)

primary\_expression = nil\_literal | bool\_literal | real\_literal | string\_literal | identifier | call\_expression;

(\* concrete \*)

nil\_literal = "nil";

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

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

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

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

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

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

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

(\*\*\* types \*\*\*)

(\* abstract \*)

type = primitive\_type | function\_type;

primitive\_type = nil\_type | bool\_type | real\_type | string\_type;

(\* concrete \*)

nil\_type = "Nil";

bool\_type = "Bool";

real\_type = "Real";

string\_type = "String";

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

# Appendix C – Operator Precedence Table

The operator precedence of *MattyLang v1.0* is defined as follows:

|  |  |
| --- | --- |
| Precedence | Operators |
| 0 (Lowest) | Logical Operators: &&, || |
| 1 | Relational Operators: <, >, <=, >=, ==, != |
| 2 | Arithmetic Operators (Additive): +, - |
| 3 | Arithmetic Operators (Multiplicative): \*, /, % |
| 4 (Highest) | Unary Operators: !, - |

# Appendix D – 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/\_\_init\_\_.py | Compiler API front-end |
| 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 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 scoping |
| mattylang/visitor.py | Base class for visitors, whose methods visit all chilren of a node |