# Projecting Through an Imaginary Programming Language

kat : phasewalk1

February 2023

**Abstract**

The Scythe programming language is a mysterious tool for the modern systems programmer, offering a combination of baremetal control and memory safety. With its advanced borrow checker and minimalistic design, the language delivers a lean and efficient standard toolchain that is both powerful and free of bloat. The naming conventions in its standard libraries and types may seem esoteric, but for those with the skill to wield them, the language opens up a world of potential. Despite being in its early stages of development, the language holds great promise, embodying the spirit of gnu and the wonder of hacking.

# 1 The Scythe Programming Language

Scythe is an LLVM-compiled language, which means Scythe benefits from LLVMs optimizer. In its early stages, Scythe attempts to maintain an execution speed similar to Rust and the C family. This is achieved primarily by performing no garbage collection (like C), but instead, borrow checking at compile-time. Scythe lets its influences shine through in every line of code.

## 1.1 Design

Scythe follows a strictly typed, imperative programming model that can feel much like a member of the C family or something akin to Rust. However, Scythe has a stricter scope system to give the programmer fine control over what code is being brought into scope, encouraging the strip of bloat at every turn. Initially, Scythe doesn't even include debugging symbols in its binaries.

# 2 Specification

## 2.1 Bridges, Worlds

In Scythe, modules or scopes can be in one of two states:

1. world
2. bridge

Bridges and worlds are module-level and root-level scopes, respectively. In writing a binary application or library, within your root file you would open a *world*. When defining modules within your application or library, you would open *bridges* in separate .scythe files and link them together. The standard syntax for opening a bridge or world is,

```
open world @x where {}
open bridge @y where {}
```

where $x$ and $y$ are world identifiers, i.e., programmer defined names that identify them in a world.

### 2.1.1 Hello, world

Worlds are how Scythe packages its code, whether building a binary or a library, a Scythe codebase is to be viewed as its own world. Worlds can pull other worlds into them, though it isn't recommended unless its either a small world, or your world needs everything that's packaged in the other; this gives the programmer more flexibility to control exactly what code is being linked into theirs and many creative keywords to match it.

### 2.1.2 Wait, It's All Bridges?

Yes, it always has been. Scythe bridges are scopes in which all functions and interfaces are to be defined, thus, worlds are merely portals which are connected to many bridges. This evokes a paradigm that prevents a program from defining any function or interface within a world, and only allows worlds to process procedures from the bridges which they mount to.

### 2.1.3 Using Bridge and World Modifiers

Scythe is a verbose language, and it has a handful of bridge and world specific keywords for mutating the state of the scope. The most commonly used one, and one you'll become familiar with, is `takeallfb`. `takeallfb` takes all items from a bridge and brings them into scope. There are many variants of this keyword, e.g., `takeallfw` takes all items from a world (not recommended), and `take` takes one item from a bridge or a whole bridge. There is also `takemask` which will make sense shortly.

Masks are shorthand for using *something* as *something else* and can be thought of as aliases. Thus, `takemask` takes an item or bridge, and masks it to the argument received. Below are some examples of bridge and world modifiers being used.

```
open bridge @client where {
  // the intermediate 'limbo' between bridges
  pragma pool {
  // imports "rts" into this scope
  take       std::talisman::rts
  // imports "def::*" into this scope
  takeallfb std::talisman::def;
  // masks "std::io::filesystem" to "fs"
  takemask   fs = std::io::filesystem;
  takemask   ioerr = std::io::Error;
  mask       pathbuf = fs::path::PathBuf;
  }

  // the pragma pool can be cleaned when a bridge
  // is no longer needed

   // removes "rts" from scope
  pragma pool clean { @rts };
  // removes the "pathbuf" mask
  pragma pool cleanmask { @pathbuf };
  // cleans scope of "def::*"
  pragma pool closebridge { @rts };
}
```

## 2.2 Functions

Scythe functions behave just like functions in any other language, however, can be defined in a more verbose manner that allows for cool behavior not present in other languages. With Scythes emphasis on bridge/world modularity, it follows that functions each must too have their own visibility specifiers. These are `pub`, `guarded` and `chained`; analogous to C++'s public, protected, and private, respectively. Imagine we are building an RPC server, and one of our server methods needs to hash an input from the client, our function prototype may look like this:

```
open bridge @methods where {
  pragma pool {
    takemask hasher = std::hash::Sha256Hasher;
  }

  // our function declaration/prototype
  chained func hash(
    str input,
  ) catches maybe<str> or voidptr {};

  // our function definition
  mkfunc hash(str input,) catches maybe<str> or voidptr {
    give h: mut hasher = hasher::new();
    try h::update(input) or throw voidptr;
    trygiveor h::finalize() to hash_final or throw voidptr;
    return hash_final;
  }
}
```