Will Lillis
DATE HERE

CONTEXT

This template was made to stylize the reports I wrote for the **Forta Foundation**.

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Template creation	2023/07/31	Apehex
1.0	Complete template	2023/10/04	Apehex
1.1	Various improvements	2024/01/11	Apehex

CONTACTS

CONTACT	MAIL	MORE
Will Lillis	**will.lillis@gmail.com**	**Github: WillLillis**

CREDITS

The template started from the **Legrand Orange template**.

The cover page started from the templates in **Latexdraw**.

The syntax highlighting for Solidity has been made by **Sergei Tikhomirov**.

PART 1

1. INTRODUCTION

1.0.1. Introduction

I started my programming journey much later than a lot of my peers. Coding and computers were scary topics best left to child prodigies and other geniuses, so why even both I reasoned. Nevertheless, following my first year of undergrad I landed an internship with my school's Physics department. Little did I know the internship was about 1% physics, and 99% learning C++. My professor worked in low-energy nuclear physics, and his group's experiments generated large amounts of data that needed to be processed. My first introduction with "real" coding was with C++, doing large-scale analysis. I spent my summer switching between online documentation and gedit, the text editor he showed me on my first day. Following this internship, I started to learn a bit of C, and was happy when I found out I could change the colors in Notepad++.

Eventually a friend took pity on me and pointed me out to Visual Studio. I cannot understate how mindblowing things like autocomplete, jump to definition, and intelligent syntax highlighting was to me after months of default Notepad++. I didn't have to spend time searching through a code base to find a type definition, didn't have to run gcc to find out I had forgotten a semilolon on line 64, etc. I wondered how these features worked, but understanding seemed beyond my grasp. If I'm struggling to even write code, then code that can "understand" other code has to be on some elevated plane of complexity.

I continued on through my classes, learning lots of theory, but not coding very much. Eventually I stumbled upon The Primeagen by accident on YouTube one day. It's hard to put into words how much he changed my outlook on programming and CS for the better. After months of listening to him shill for Neovim and Rust, I decided to spend a summer learning the language and editor. These things scared me, but I had finally gotten over this fear and just decided to try my best and see where I ended up.

I *cannot* understate what a great decision this was. Coding has never been more fun. After reading through the Rust book, I set out to complete the 2022 Advent of Code calendar. Working through those exercises (and sometimes referring to Reddit for help), I slowly figured things out. After a few weeks, I was comfortable enough to start making small modifications to the Lua configuration I had copied from Prime's setup video. I finished Advent of Code, and started looking for a larger project to work on.

I eventually stumbled onto **`asm-lsp`**, an open-source assembly LSP written in Rust. The project had hover support for instructions, but that was about it. I found a small `// TODO` comment in the source code, and decide to take a look. After an hour of hopping around the project with the help of the Rust LSP, I thought I had an idea of how to solve the problem. I opened a pull request, and to my surprise it was accepted. I started digging into the code a bit more, and found another thing to fix. And then I found a feature to add. And then another. And another. Over the course of a few months, I got very comfortable with the LSP's code, and in turn also found myself fairly comfortable with the LSP protocol. Working on the project had demysitified the "magic" code that understood code, and I found myself enjoying the rabbit hole I had unwitteningly dived down.

This learning journey was incredibly valuable to me, and I'd like to help others discover this magic.

1.0.2. What is this book?

As the title suggests, this book walks you through building an LSP server for x86-64 assembly. It won't be the most complete LSP server, it won't be the cleanest, but it will be *yours*. We'll start with a basic scaffold and build up from there. So anyways, onto the burning question:

"Why Assembly? Nobody actually codes in that anymore."

Fair enough, almost nobody actually hand rolls assembly anymore. Those damned compiler people have done too good of a job! (If you're interested in compilers though, I can't reccomend Thorsten Ball's **"Writing an Interpreter in Go"** and **"Writing a Compiler in Go"** enough. Both are incredibly well done and helped motivate me to write this book.) But in all seriousness, I've chosen assembly for its *simplicity*. Assembly is simple enough that the majority of one's effort can be spent learning the main item, LSPs! Assembly's flat structure makes things just simple enough so that lots of features can be doable without weeks of studying a specific language's features. The obvious alternative is to use a toy language, which in my opinion sucks all the fun out. Building things for the sake of building is great, but creating something useful along the way is better! All you really have to know is that there are instructions and registers. Instructions tell the computer to do something and typically take 0 to 2 arguments. Registers are like tiny variables for the CPU. They vary in size, usually a nice power of 2 from 16 bits and up. Finally there are assembler directives, which basically just tell the assembler to do something, but aren't translated to machine code like instructions are. To be more concrete, here's a simple hello world program written for the GNU Assembler (GAS).

```
# gcc -nostdlib -nostartfiles -nodefaultlibs -static hello_lsp.s -o
    hello_lsp && ./hello_lsp

.data

.equ    SYSCALL_WRITE, 1
.equ    SYSCALL_EXIT, 60
.equ    STDOUT, 1
.equ    EXIT_SUCCESS, 0

message:
    .asciz "Hello, LSP!\n" # Our message to print

.equ MSG_LEN, . - message # The length of our message

.global _start

.text

_start:
    # write (1, msg, len(msg))
    mov $SYSCALL_WRITE, %rax
    mov $STDOUT, %rdi
    mov $message, %rsi
    mov $MSG_LEN, %rdx
    syscall

    # exit(0)
```

```
28      mov $SYSCALL_EXIT, %rax
29      xor %rdi, %rdi
30      syscall
```

Without going into too much detail, the `.data` directive simply tells the assembler to assemble the following statements into a data subsection. Underneath that, `.equ` works essentially the same as the C preprocessor's `#define`. Finally `.asciz` creates a null terminated C string. We then grab the length of the message. Some data gets moved around so we can make the `write` and `exit` system calls, and that's about it! The specifics can vary depending on architecture and assembler you're using, but most assembly is fairly similar in structure.

We'll start with some general LSP information and project setup. While I won't skip any steps, I'll do my best to speed through the non-LSP parts to keep things interesting. While there's lots to be learned about things like XML parsing, serialization, deserialization, and JSON RPC, I'm not the right person and this isn't the right book to teach those things. After these initial steps, we'll have a working program that can connect to an LSP client, load some relevant information from the disk...and that's about it. But don't despair! The very next thing on our list is to add hover support for instructions and registers. After this feature is in place, we'll sidetrack briefly to add some configuration options for our users (of which there will be many thousands, so we can also practice asking the product manager "but how will this scale?"). Next we'll add autocomplete!

Doing so will require a couple different steps, but this is when things truly start to get fun. When I first added autocomplete to asm-lsp, things really started to click. After autocomplete, we'll add a bunch of cool (but relatively siloed) capabilities.

2. GETTING STARTED

All of the finished code for each chapter can be found in the **book's repo**. With that being said, let's get started!

2.0.1. Dependencies

As with nearly any other Rust project, we can start by creating a new project with cargo: `cargo new assembly-lsp`. We'll add some dependencies next:

- `cargo add crossbeam-channel`
  - Handles sending and receiving data across channels so our LSP server can talk to our editor's LSP client
- `cargo add flexi_logger`
  - Allows for some easy structured logging over `stderr`.
- `cargo add log`
  - Another logging crate we'll use
- `cargo add lsp-server`
  - Handles the majority of LSP-related things for us. This crate lets us focus on features, rather than implementing a spec.
- `cargo add lsp-types`
  - The LSP spec includes a fair number of type defitinitions, which are (unfortunately) in Typescript. This crate provides Rust versions of the spec's types that adhere to the spec.
- `cargo add serde --features derive`
  - This amazing crate is part of nearly every Rust project, including ours too.
- `cargo add serde_json`
  - LSP servers and clients communicate over JSON RPC. Naturally this crate will be of use.

2.0.2. Starting Code

Now that the scaffolding is in place, we can finally get to coding. A great template can be found in the rust-analyzer (the absolutely excellent Rust LSP) **repo**. Their example starts out with some code for the go to definition capability. We'll get there eventually, but our starting point is different. Here's the stripped down code:

```
1  use log::info;
2  use lsp_server::{Connection, ExtractError, Message, Request, RequestId};
3  use lsp_types::{InitializeParams, ServerCapabilities};
4
5  fn main() -> anyhow::Result<()> {
6      // Set up logging. Because `stdio_transport` gets a lock on stdout
       and stdin, we must have our
7      // logging only write out to stderr.
8      flexi_logger::Logger::try_with_str("info")?.start()?;
9
10     info!("Starting assembly-lsp");
11
```

```
    // Create the transport. Includes the stdio (stdin and stdout) versions but this could
    // also be implemented to use sockets or HTTP.
    let (connection, io_threads) = Connection::stdio();

    // Run the server and wait for the two threads to end (typically by trigger LSP Exit event).
    let server_capabilities = serde_json::to_value(&ServerCapabilities {
        ..Default::default()
    })
    .unwrap();

    let initialization_params = connection.initialize(server_capabilities)?;

    main_loop(connection, initialization_params)?;
    io_threads.join()?;

    // Shut down gracefully.
    info!("Shutting down assembly-lsp");
    Ok(())
}

fn main_loop(connection: Connection, params: serde_json::Value) -> anyhow::Result<()> {
    let _params: InitializeParams = serde_json::from_value(params).unwrap();
    info!("Entering main loop");
    for msg in &connection.receiver {
        eprintln!("got msg: {msg:?}");
        match msg {
            Message::Request(req) => {
                if connection.handle_shutdown(&req)? {
                    return Ok(());
                }
                info!("Got request: {req:?}");
            }
            Message::Response(resp) => {
                info!("Got response: {resp:?}");
            }
            Message::Notification(notif) => {
                info!("Got notification: {notif:?}");
            }
        }
    }
    Ok(())
}

fn cast_req<R>(req: Request) -> Result<(RequestId, R::Params), ExtractError<Request>>
where
    R: lsp_types::request::Request,
    R::Params: serde::de::DeserializeOwned,
```

```
{
    req.extract(R::METHOD)
}
```

There's nothing too crazy here, but it'll help to unpack everything briefly. Let's start from the top! The first thing we do is set up our logging. As the LSP protocol uses `stdout` for client-server communication, we configure our logger to use `stderr`. Next we use the `lsp-server` crate to open a connection between the LSP server (our program) and an LSP Client (likely a text editor).

Next we have an object to specify our server capabilities to the client. If you've taken a look at the **LSP spec**, you probably noticed a large number of capabilities. The great thing about the LSP spec is that we only have to implement the capabilities we want to. From the **docs**:

> "Not every language server can support all features defined by the protocol. LSP therefore provides 'capabilities'. A capability groups a set of language features."

The capabilities are all optional, and if look at the `lsp-types` crate, we can see that every capability-related type is of type `Option<T>`. As the default value for any `Option` type is `None`, we can safely specify the capabilities we care about, and ignore all the others.

With our (currently empty) capabilities defined via `server_capabilities`, we communicate them via the 'initialize' request and receive client's capabilities back in turn. With this handshake out of the way, we then enter the meat of the program, which is just a simple event loop to receive and respond to requests from the client. Rust's pattern matching really shines here, letting us easily break down our server's actions by the type of message its received. Our event loop doesn't do anything besides print out the messages it receives. At this point it would probably be useful to test things out and make sure the program is behaving as expected. While we'd normally just test the things by running the LSP with our editor of choice, for pedagogical purposes it can be useful to see how things work manually. If we run our server from the command line, (with a little help from `sed` we can act as the LSP client and manually enter JSON RPC messages. Starting with

```
    sed -u -re 's/^(.*)$/\1\r/' | cargo run
```

we should see our program build and then start. Entering

```
Content-Length: 85

{"jsonrpc": "2.0", "method": "initialize", "id": 1, "params": {"capabilities":
    {}}}
```

followed by

```
Content-Length: 59

{"jsonrpc": "2.0", "method": "initialized", "params": {}}
```

will give our server enough information to enter into it's event loop. This isn't a good way to interact with or test the, but it's helpful to peel back the curtain at times and see that this isn't magic. It really is just JSON RPC over `stdout`. We can wrap things up by sending shutdown and exit requests.

```
Content-Length: 67

{"jsonrpc": "2.0", "method": "shutdown", "id": 3, "params": null}
```

followed by

```
Content-Length: 54

{"jsonrpc": "2.0", "method": "exit", "params": null}
```

All of this together:

```
$ sed -u -re 's/^(.*)$/\1\r/' | cargo run

Compiling assembly-lsp v0.1.0 (/home/lillis/projects/LSPBook/assembly-lsp)
warning: function `cast_req` is never used
  --> src/main.rs:55:4
   |
55 | fn cast_req<R>(req: Request) -> Result<(RequestId, R::Params),
    ExtractError<Request>>
   |    ^^^^^^^^
   |
   = note: `#[warn(dead_code)]` on by default

warning: `assembly-lsp` (bin "assembly-lsp") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 1.30s
     Running `target/debug/assembly-lsp`
INFO [assembly_lsp] Starting assembly-lsp
Content-Length: 85

{"jsonrpc": "2.0", "method": "initialize", "id": 1, "params": {"capabilities":
    {}}}
Content-Length: 53

{"jsonrpc":"2.0","id":1,"result":{"capabilities":{}}}Content-Length: 59

{"jsonrpc": "2.0", "method": "initialized", "params": {}}
INFO [assembly_lsp] Entering main loop
Content-Length: 67

{"jsonrpc": "2.0", "method": "shutdown", "id": 3, "params": null}
got msg: Request(Request { id: RequestId(I32(3)), method: "shutdown", params:
    Null })
Content-Length: 38

{"jsonrpc":"2.0","id":3,"result":null}Content-Length: 54

{"jsonrpc": "2.0", "method": "exit", "params": null}
INFO [assembly_lsp] Shutting down assembly-lsp
$
```

Finally, underneath the main loop we have a small helper function `cast_req`. Depending on your experience level with Rust, this function signature may look a little scary, but I promise it's not. Per the LSP spec, the parameters passed along with each request all have distinct types. The trait bounds used in the function's signature simply allows it to take in a request and return the correct parameter type.

2.0.3. LSP to Editor Connection

Now that most of the boilerplate-y code is in place, let's connect our server to our editor. This process can vary greatly from editor to editor.

I personally use Neovim, and would highly reccomend it to anyone interested in creating their own **Personal Development Environment**. However with it being a very personalized editor, methods for attaching a new LSP can cary greatly depending on one's config. For the purposes of this book however, we want to focus on LSPs, not finding the right ~~incantation~~ set of configuration options to make the two processes connect. I've gone through this process for a few editors, and found Sublime's setup to be the easiest *by far*. I would highly encourage you to set up the LSP in your editor of choice. However, in the interest of having some guaranteed way to test things, I'll go through the sublime setup below. I've included the specific steps I followed on my Ubuntu system, but if that doesn't work I've also provided links to the relevant documentation.

1. Install **Sublime Text** if you don't already have it
   - For me, this was just `sudo apt-get install sublime-text`
2. Install Sublime's **LSP** client
   - Open Sublime
   - Open the command pallet (default is Control+Shift+p)
   - Run `Package Control: Install Package`
   - Select `LSP`
3. Ensure the executable produced for our program can be found on the system `PATH`
4. Configure Sublime to work with our LSP
   - Open the command pallet (default is Control+Shift+p)
   - Run `Package Control: Install Package`
   - Select the **x86 and x86_64 Assembly** package
   - Open `Preferences > Package Settings > LSP > Settings` and add the "`assembly-lsp`" client configuration to the "`clients`":

```
{
    "clients": {
        "assembly-lsp": {
            "enabled": true,
            "command": ["assembly-lsp"],
            "selector": "source.asm | source.assembly"
        }
    }
}
```

...And that should be it! We can make sure things are working by opening the project's test file in Sublime and checking the logs (`Tools > LSP > Toggle Log Panel`). We should see the same logs we added earlier, as well as the JSON messages exchanged as part of the protocol. That's all the setup we'll do for now, let's get started on our first feature

3. HOVER SUPPORT

The time is now! We're finally adding a real feature to our LSP! We'll start by adding hover support for instructions. A few things need to happen first though, before we see that sweet sweet hover documentation window. First we need some source of information for our instructions. This source needs to be in a fairly organized, regular format fit for parsing. After finding such a source, we then need to write some code to parse and load it into our program. This will require some parsing code (duh), as well as some data structures to represent our information once it's loaded from disk. Finally, we need to trigger every time a hover request comes in from the client, check our relevant data structures, and return any results we find in the right format. Without further ado, let's get started.

3.0.1. Finding an Information Source

The first thing we need is find some kind of data source for the x86_64 instructions. It should be well organized, reasonably convertible to some in-memory data structure, and hopefully actively maintained (many active assembly languages receive regular updates). Lucky for us, such a data source already exists! The open source **Opcodes** repo conveniently has both the x86 and x86_64 instruction sets in an xml file. Let's pull the appropriate file into our project:

```
mkdir opcodes && cd opcodes
curl https://github.com/Maratyszcza/Opcodes/blob/master/opcodes/x86.xml --
    output x86.xml
```

Our project should look two directories now (ignoring the `target` build directory).

```
assembly-lsp
|--Cargo.lock
|--Cargo.toml
|--opcodes
|  |--x86.xml
|--src
   |--main.rs
```

Let's take a look inside that new file!

```
<?xml version='1.0' encoding='utf-8'?>
<InstructionSet name="x86">
  <Instruction name="AAA" summary="ASCII Adjust After Addition">
    <InstructionForm gas-name="aaa" go-name="AAA">
      <Encoding>
        <Opcode byte="37"/>
      </Encoding>
    </InstructionForm>
  </Instruction>
  <Instruction name="AAD" summary="ASCII Adjust AX Before Division">
    <InstructionForm gas-name="aad" go-name="AAD">
      <Encoding>
        <Opcode byte="D5"/>
        <Opcode byte="0A"/>
      </Encoding>
    </InstructionForm>
    <InstructionForm gas-name="aad" go-name="AAD">
      <Operand type="imm8"/>
```

```
      <Encoding>
        <Opcode byte="D5"/>
        <Immediate size="1" value="#0"/>
      </Encoding>
    </InstructionForm>
  </Instruction>
  <Instruction name="AADD" summary="Atomically ADD">
  ...
```

It looks like each instruction has its name stored along with a short summary. Nested inside each `<Instruction ...` there also appears to be one or more instruction forms for the GNU Assembler (GAS) or the Go Assembler. Each form has some more information included, such as the opcodes for that form, the different arguments it can take, etc. How are we going to represent this data inside our program?

3.0.2. Data Structures and Parsing

This calls for a new file! Let's create `src/instruction.rs` and start laying out our data. Here's a start...

```
pub struct Instruction {
    name: String,
    summary: String,
    ...
}
```

Those instruction forms seem fairly important, so let's create a struct to represent them and then store a vector of forms inside each instruction. Given that there are opcode files for both x86 and x86_64, we should probably put some kind of architecture information in there as well. Finally, we'll also derive some common methods on these to make our lives easier down the road.

```
#[derive(Debug, Clone)]
pub struct Instruction {
    name: String,
    summary: String,
    arch: Option<Arch>,
    forms: Vec<InstructionForm>,
}

#[derive(Debug, Clone)]
pub struct InstructionForm {
    gas_name: Option<String>,
    go_name: Option<String>,
    encoding: String,
    operands: Vec<Operand>
}

#[derive(Debug, Clone)]
pub struct Operand {
    op_type: OperandType,
    input: bool,
    output: bool,
```

```
}

#[allow(non_camel_case_types)]
#[derive(Debug, Clone, Copy)]
pub enum Arch {
    x86,
    x86_64,
}
```

Writing out all of the variants for the `OperandType` enum looks really annoying and tedious...luckily I've done it for you! For the sake of brevity, I'll also take this time to introduce some new crates (`cargo add strum` and `cargo add strum_macros` when you have a minutes) to our project. We'll use these dependencies to great effect while parsing and deserializing the `x86.xml` file. Some operand types conflict with Rust's naming rules for its enum variants, so in those cases I've changed the variant name to something reasonably close, and then included the `#[strum(serialize = "<ActualOpTypeHere>")]` macro to make the translation possible. We'll also apply these helpers to our `Arch` enum.

```
use strum_macros::{AsRefStr, EnumString};

...

#[allow(non_camel_case_types)]
#[derive(Debug, Clone, Copy, EnumString, AsRefStr)]
pub enum Arch {
    x86,
    x86_64,
}

#[allow(non_camel_case_types)]
#[derive(Debug, Clone, EnumString, AsRefStr)]
pub enum OperandType {
    #[strum(serialize = "1")]
    _1,
    #[strum(serialize = "3")]
    _3,
    imm4,
    imm8,
    imm16,
    imm32,
    imm64,
    al,
    cl,
    r8,
    r8l,
    ax,
    r16,
    r16l,
    eax,
    r32,
    r32l,
    rax,
    r64,
```

```
    mm,
    xmm0,
    xmm,
    #[strum(serialize = "xmm{k}")]
    xmm_k,
    #[strum(serialize = "xmm{k}{z}")]
    xmm_k_z,
    ymm,
    #[strum(serialize = "ymm{k}")]
    ymm_k,
    #[strum(serialize = "ymm{k}{z}")]
    ymm_k_z,
    zmm,
    #[strum(serialize = "zmm{k}")]
    zmm_k,
    #[strum(serialize = "zmm{k}{z}")]
    zmm_k_z,
    k,
    #[strum(serialize = "k{k}")]
    k_k,
    moffs32,
    moffs64,
    m,
    m8,
    m16,
    #[strum(serialize = "m16{k}")]
    m16_k,
    #[strum(serialize = "m16{k}{z}")]
    m16_k_z,
    m32,
    #[strum(serialize = "m32{k}")]
    m32_k,
    #[strum(serialize = "m32{k}{z}")]
    m32_k_z,
    #[strum(serialize = "m32/m16bcst")]
    m32_m16bcst,
    m64,
    #[strum(serialize = "m64{k}")]
    m64_k,
    #[strum(serialize = "m64{k}{z}")]
    m64_k_z,
    #[strum(serialize = "m64/m16bcst")]
    m64_m16bcst,
    m128,
    #[strum(serialize = "m128{k}")]
    m128_k,
    #[strum(serialize = "m128{k}{z}")]
    m128_k_z,
    m256,
    #[strum(serialize = "m256{k}")]
    m256_k,
    #[strum(serialize = "m256{k}{z}")]
    m256_k_z,
```

```
    m512,
    #[strum(serialize = "m512{k}")]
    m512_k,
    #[strum(serialize = "m512{k}{z}")]
    m512_k_z,
    #[strum(serialize = "m64/m32bcst")]
    m64_m32bcst,
    #[strum(serialize = "m128/m32bcst")]
    m128_m32bcst,
    #[strum(serialize = "m256/m32bcst")]
    m256_m32bcst,
    #[strum(serialize = "m512/m32bcst")]
    m512_m32bcst,
    #[strum(serialize = "m128/m16bcst")]
    m128_m16bcst,
    #[strum(serialize = "m128/m64bcst")]
    m128_m64bcst,
    #[strum(serialize = "m256/m16bcst")]
    m256_m16bcst,
    #[strum(serialize = "m256/m64bcst")]
    m256_m64bcst,
    #[strum(serialize = "m512/m16bcst")]
    m512_m16bcst,
    #[strum(serialize = "m512/m64bcst")]
    m512_m64bcst,
    vm32x,
    #[strum(serialize = "vm32x{k}")]
    vm32x_k,
    vm64x,
    #[strum(serialize = "vm64x{k}")]
    vm64xk,
    vm32y,
    #[strum(serialize = "vm32y{k}")]
    vm32yk_,
    vm64y,
    #[strum(serialize = "vm64y{k}")]
    vm64y_k,
    vm32z,
    #[strum(serialize = "vm32z{k}")]
    vm32z_k,
    vm64z,
    #[strum(serialize = "vm64z{k}")]
    vm64z_k,
    rel8,
    rel32,
    #[strum(serialize = "{er}")]
    er,
    #[strum(serialize = "{sae}")]
    sae,
    sibmem,
    tmm,
}
```

3.0.3. Hover Requests

3.0.4. Cleanup

DOLOR SIT AMET

4. HENDRERIT SAPIEN

4.1. SED ULTRICES

4.1.1. Donec pellentesque

Tempus ipsum, vitae condimentum nisi efficitur id. In velit mauris, auctor eget sapien nec, viverra mattis neque. Nunc vel commodo nunc, eget cursus ex.

1. Nunc maximus consequat tristique.
2. Praesent luctus ex aliquam rhoncus consectetur.
3. Suspendisse mattis velit ante, vitae mollis est pharetra a.
4. Duis tincidunt, urna id auctor imperdiet, odio dui ullamcorper nisi.
5. Integer tincidunt enim vitae nulla iaculis, in varius metus blandit.
6. Nunc quam arcu, fermentum non dapibus condimentum, condimentum eget nunc.

4.1.2. Euismod sodales

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

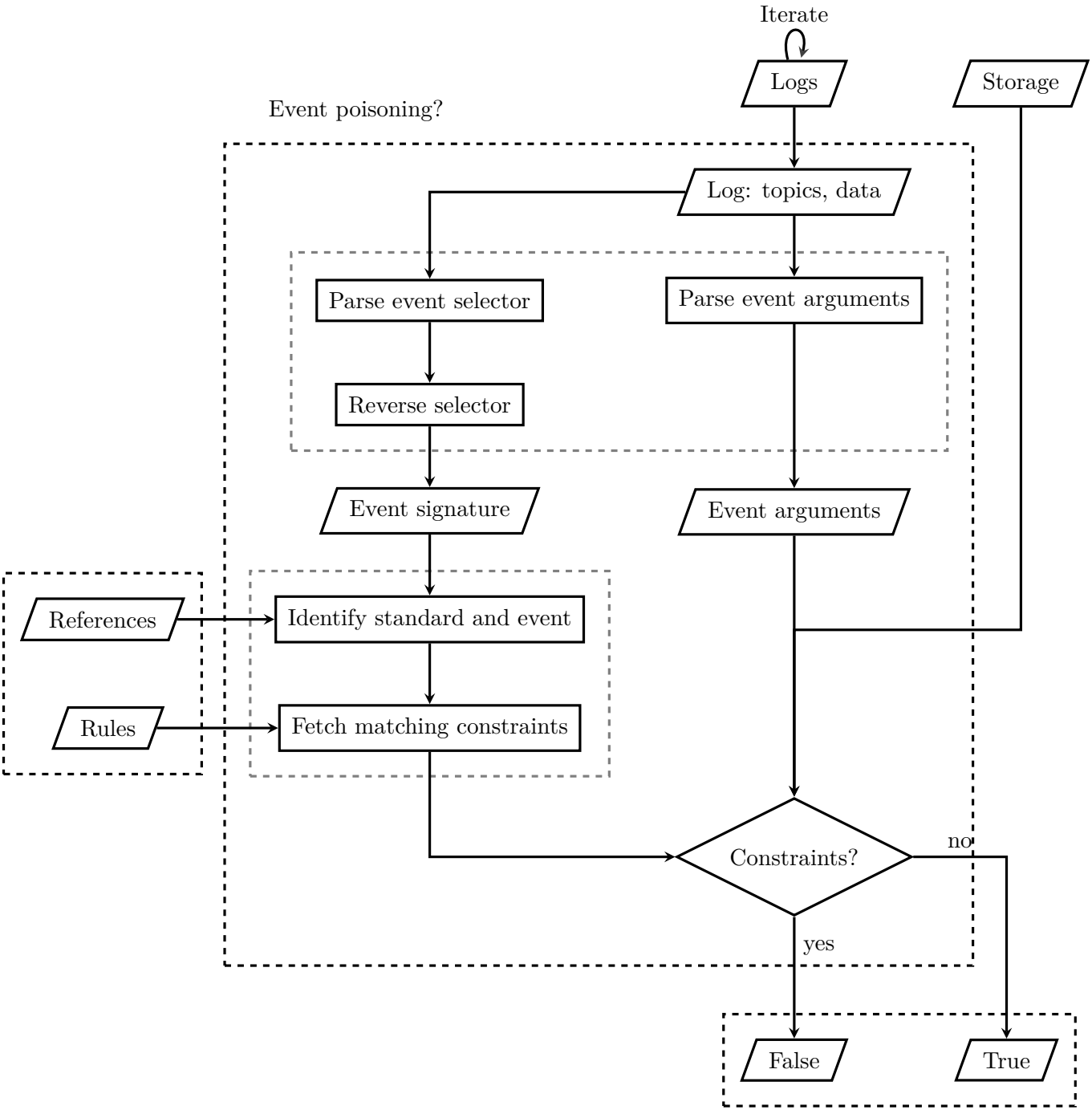
4.1.3. Pellentesque a nulla

```
1  contract KingOfTheHill is Ownable {
2      address public owner; // different from the owner in Ownable
3
4      function () public payable {
5          if(msg.value > jackpot) owner = msg.sender; // local owner
6          jackpot += msg.value;
7      }
8      function takeAll () public onlyOwner { // contract creator
9          msg.sender.transfer(this.balance);
10         jackpot = 0;
11     }
12 }
```

4.2. PHASELLUS TRISTIQUE

Lacus ac turpis semper fringilla. Mauris fermentum varius neque, vel congue elit tempus quis. Aliquam a nunc in nunc consectetur hendrerit ut quis felis.

Quisque id dui magna. Curabitur posuere nisl at magna vehicula aliquam 4.1. Duis et suscipit felis. Mauris non justo quis diam gravida placerat condimentum ut sapien. Praesent imperdiet libero id est tincidunt ullamcorper.



Maecenas hendrerit congue faucibus

APPENDICES

5. VESTIBULUM COMMODO

INTEGER SEMPER DICTUM TELLUS

Neque eu cursus faucibus, ipsum magna tincidunt dui, vel scelerisque urna nibh ut nisl:

Duis sit amet mattis magna.
Curabitur sit amet fermentum mi.
Morbi convallis purus eu fermentum accumsan, mauris felis consequat ipsum.
Ut sollicitudin sit amet tellus et mollis.

A PORTTITOR ORCI FAUCIBUS SIT AMET

Vivamus et sapien vitae lacus ornare suscipit vitae id mauris:

Vivamus in dui arcu.
Morbi vitae dolor libero.
Tellus est, pellentesque at ultrices non, consectetur sit amet augue.
Suspendisse elementum mollis nisl at aliquam.

6. SOLLICITUDIN

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

// github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.0.0/contracts/token/ERC20/IERC20.sol
interface IERC20 {
    function totalSupply() external view returns (uint);

    function balanceOf(address account) external view returns (uint);

    function transfer(address recipient, uint amount) external returns (bool);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint amount) external returns (bool);

    function transferFrom(
        address sender,
        address recipient,
        uint amount
    ) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);
}
```

```json
{
    "blockHash": "0xfff25d13f3e37982e6a380771a52ea79d60d4592fb13f24a157a96bd48cb823a",
    "blockNumber": "0x011367d1",
    "from": "0x7d9bc45a9abda926a7ce63f78759dbfa9ed72e26",
    "gas": "0x01724b",
    "gasPrice": "0x030305bec8",
    "maxFeePerGas": "0x03f6b8baa1",
    "maxPriorityFeePerGas": "0x05f5e100",
    "hash": "0xe0a98402cb9b9536b4b308458ba46f23d41a51dfd6ee4102c118230a44529cbd",
    "input": "0xa9059cbb000000000000000000000000e897c0f9443785f8d4f0fa6e...",
    "nonce": "0x17",
    "to": "0xdac17f958d2ee523a2206206994597c13d831ec7",
    "transactionIndex": "0xd5",
    "value": "0x00",
    "type": "0x02",
    "accessList": "0x",
    "chainId": "0x01",
    "v": "0x01",
    "r": "0x66ffefa83e6ce55d11ee5d9eb5e0534b9b8579d2e354f8ea72d8f91250fc5893",
    "s": "0x328dc78a37989de1c52bd4b5f24c1a7f9796981e6eb8ac520a5945b06888fef4",
    "yParity": "0x01"
}
```

7. ETIAM FACILISIS

7.1. IPSUM PRIMIS

In faucibus orci luctus et ultrices posuere cubilia curae; Aliquam nunc ipsum, sollicitudin ut tempus consectetur, fermentum at lectus.

Nullam molestie viverra **??** augue sit amet gravida.

7.1.1. Mauris pellentesque

Massa sagittis malesuada

Symbol	Unit	Description
g	m/s^2	nisi in mollis gravida

7.1.2. Nulla massa

Quis imperdiet

Symbol	Unit	Description
Q_p	t/h	consectetur tortor

Magna lectus

Symbol	Unit	Description
τ_a	$^\circ C$	integer mollis bibendum gravida
f_s	$/jour$	vestibulum porta urna at ex dignissim tincidunt

7.1.3. Nam vitae

Symbol	Unit	Description
X_n	m	venenatis nisi
Y_n	m	integer malesuada eu ligula nec pharetra
Z_n	m	fusce non elit lectus

LIST OF FIGURES

LIST OF TABLES