

362 SOFTWARE ENGINEERING GROUP PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Making Smart Contracts Safer

Authors:

Aurel Bílý

Catalin Craciun

Calin Farcas

Yicheng Luo

Constantin Mueller

Niklas Vangerow

Supervisor:

Susan Eisenbach

Date: January 4, 2019

Contents

1	Executive Summary	1
2	Motivations, Objectives, and Achievements	2
2.1	Motivation	2
2.2	Objectives	2
2.3	Achievements	3
3	Project Management	4
3.1	Planning	4
3.2	Organisation	6
3.2.1	Process	6
3.2.2	Project Board	6
3.2.3	Review Process	8
3.2.4	Communication	9
3.3	Task Allocation and Roles	10
4	Design and Implementation	11
4.1	Existing Codebase	11
4.1.1	Lexer and Parser	11
4.1.2	AST Passes	11
4.1.3	Example Files and Tests	12
4.2	Our Approach	12
4.3	Improving the Language	12
4.3.1	Asset Trait	14
4.3.2	External Calls	16
4.3.3	Adding Unit Testing	19
4.3.4	Refactoring Code Generation	21
4.3.5	General Improvements and Technical Challenges	22
4.4	IDE Integration	24
4.4.1	Language Server Protocol	24
4.4.2	Additional Repositories	25
4.4.3	Implemented Features	25
5	Evaluation	28

6	Reflections and Future extensions	30
6.1	Extensions	31
6.2	Ethics	33

Chapter 1

Executive Summary

Ethereum is an open source platform that enables developers to build and deploy decentralised applications by leveraging blockchain technology. In short, the blockchain enables a decentralised trust model due to computation being intrinsically tied to the progression of time by using cryptographic hashes. Past blocks cannot be altered and all nodes will eventually be consistent with each other. A currency, Ether, drives the network to incentivise users to keep it running. Ether is created by nodes that successfully verify blocks on the network (called mining) and can be transferred as part of a transaction on the ledger. Wei is a smaller denomination of Ether.

When a contract is deployed, it is added as a transaction to the ledger and is given an address which can be used to call its functions once mined. Since the blockchain is a segmented ledger, the contract cannot be modified and cannot be deleted. It is therefore crucial that a developer is certain of their code being correct.

Flint is a type-safe language syntactically inspired by Swift, designed for enabling developers to write safe smart contracts targeting blockchain platforms. Many attacks on Ethereum smart contracts in recent history could have been prevented if they were written in a development environment with modern verification features and with a language that disallows common preventable programmer errors.

The language and reference compiler were originally created by Franklin Schrans as part of his final-year individual project at Imperial College in 2017-2018, supervised by Prof. Susan Eisenbach [1]. The compiler was designed for compiling example code snippets but was limited in the kinds of contracts that could be compiled due to bugs and incomplete features. Our goal for this project was to further extend the Flint language to improve safety, and increase its appeal and utility for authors of smart contracts.

The appeal to a smart contract developer has not changed between our project and Franklin's. Fundamentally we stayed true to the Flint philosophy, both in the strong preference for adapting design decisions from the Swift language for programmer familiarity and in overall operational safety. We developed our language extensions in a public GitHub repository, enabling anyone to inspect and audit our code. At the end of the project we will contribute our language changes back to the original project repository to enable all Flint developers and maintainers to benefit from our work, for free.

Chapter 2

Motivations, Objectives, and Achievements

2.1 Motivation

At the moment, the vast majority of smart contracts are written in Solidity, a high-level statically-typed programming language. However, Solidity has few safety features and contracts written in Solidity have therefore been subject to attacks and bugs. Hundreds of millions of dollars worth of cryptocurrency are estimated to be at risk because of unsafe smart contracts. In order for the full potential of the blockchain technology to be used and the loss of money to be avoided, smart contracts need to become safer.

2.2 Objectives

Flint, like Solidity, is a programming language for smart contracts in Ethereum. Being type-safe and having a range of built-in security checks and mechanisms, Flint aims to be much safer than Solidity. Flint was developed as part of Franklin Schrans' thesis "Writing Safe Smart Contracts in Flint" [1] ; however, it lacked important and necessary features, such as external calls that allow a smart contract to interact with other contracts. In particular, without external calls there is no way to process Ether, so Flint would need to create its own coin. In addition, there was no ecosystem for Flint development with the exception of syntax highlighting in Vim and Atom. Our goal was to improve both aspects and in particular we wanted to:

- Develop an Asset trait that models cryptocurrencies to ensure that the state of currency in the contract is consistent with that on the blockchain
- Generate certifications that highlight compilation warnings and other possible problems in the code to improve the Flint ecosystem
- Design and implement external calls in order to be able to interact with other contracts and process Ether

2.3 Achievements

In the course of the project, we adapted those goals slightly as we came across new issues and re-prioritised changes to the compiler. We achieved the extension of the Flint language and the improvement of its ecosystem through the following changes:

- Adding a special Asset trait
- Adding a polymorphic Self type
- Producing diagnostic output that is compatible the Language Server Protocol and can be used for example in Microsoft Visual Studio
- Designing and implementing external calls
- Designing and implementing exception handling for external calls
- Setting up a unit testing framework for the Flint compiler
- Refactoring the code generation
- Correcting issues in the code base

Chapter 3

Project Management

In this section we will explain how our group organised and how we managed our work.

At the start of the project during our initial planning session, the group committed to a custom flavour of the popular ‘Kanban’ methodology where we dropped parts that did not work for our team. Our philosophy in adapting Kanban to our needs was that ‘our process should work for us – we should not work for our process’.

As our project involved contributing to an existing codebase, we also decided to spend the first week familiarising ourselves with the codebase. This was important for the whole group to contribute effectively, especially as most had no prior experience with the Swift programming language.

We spent the first week updating the open source project to run on the latest version of Swift as it had not been worked on for several months at that point. We also added a code linting tool called ‘SwiftLint’¹ to the build pipeline in order to ensure that all code is formatted consistently.

3.1 Planning

Every week we had a meeting on Mondays at 16:00, where we all came together and reflected on the work of the past week. Many of these meetings had interesting and heated discussions about matters of design and which direction to take the project in. Additionally the Monday meeting served as a meeting where we were able to plan the next week of work, such as features we would like to work on as well as tickets that needed to be completed for a feature to be complete.

On Wednesdays our group would meet with our supervisor to discuss our progress and what features we felt we should work on. From this process many of the features that we worked on which were not originally planned emerged, such as the work on unit testing. We would also use this as an opportunity to demonstrate our work from the two weeks prior and get the milestone assessment sheets signed in weeks where they were due.

Every day we had a virtual standup meetings on our #standups Slack channel. Group members were notified through a daily reminder that was set up on this

¹<https://github.com/realm/SwiftLint>

channel, and asked to write a small message about what they were working on and whether they needed any help from the rest of the group. No message was required from individuals who had not done any work since the last meeting. Initially we had these meetings only once during the day at 11:00, but at the start of Checkpoint 3 we decided to hold an additional meeting at 19:00. By holding two standup meetings we could coordinate more effectively with the schedules of all group members, as some member had lectures in the morning and therefore could not work on the project until the afternoon and vice versa. We noticed significantly higher participation in these meetings by having two each day rather than just one, as group members who did not work in the mornings would often miss the notification from the morning meeting.

Halfway through the project, at the end of the second checkpoint, we met to look at our progress and think about how we can change the process to make a success of the latter two checkpoints of the project. We used a website called ‘Retrium’ which allowed us to anonymously contribute thoughts as a set of virtual notes organised into several columns labelled ‘start’, ‘stop’, and ‘continue’ (see figure 3.1). It was in this meeting that we identified that we wanted to have multiple standups a day and that we should work harder to not get sidetracked. One issue throughout the project was that implementing some features required the investment of a significant amount of time to build the necessary foundations.

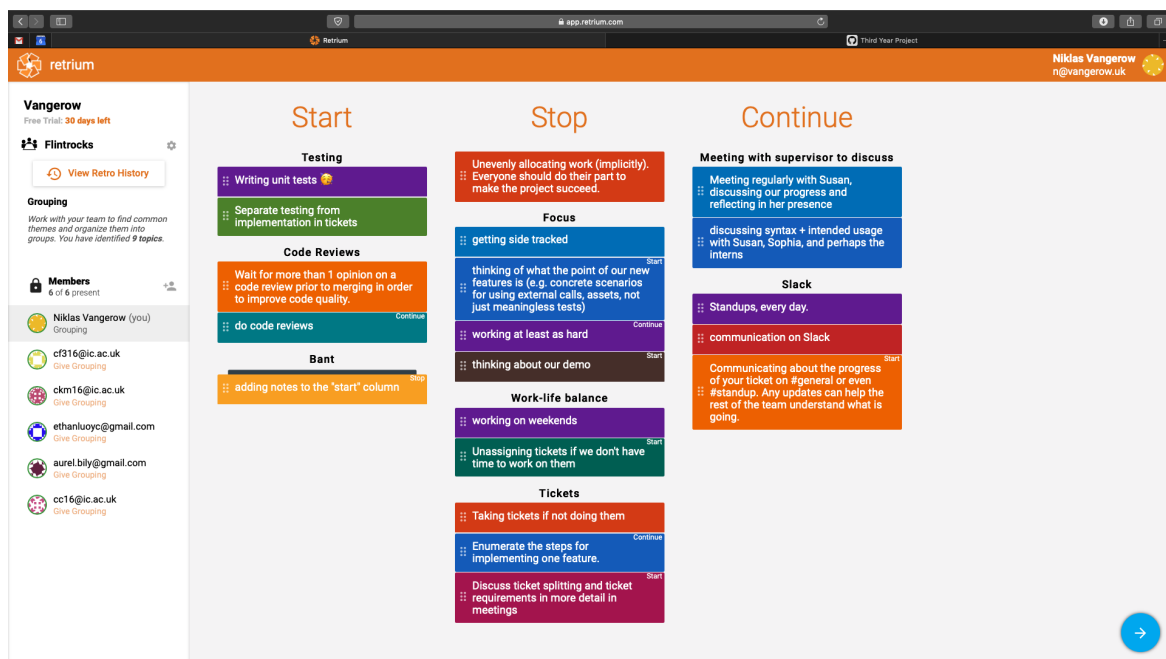


Figure 3.1: Our Retrium board showing all suggestions made by the team.

3.2 Organisation

3.2.1 Process

We chose Kanban as we wanted the flexibility of taking up new work as it happened while at the same time being aware of the work that is still pending. Our version of Kanban removed the WIP limit and instead enforced a softer limit that any one person could only be working on one ticket at any time. This kept the number of concurrent tickets low, but as the WIP limit was 6 there was less overlap and thus less need for individuals to pair.

We chose not to adopt a Scrum methodology as we deemed it to be too inflexible for our individual schedules due to vastly disparate course choices. Scrum would have forced long meetings committing to specific pieces of work ahead of time when the scope of our project was constantly changing.

3.2.2 Project Board

To give every team member the ability to see the progress of the project at a glance and indicate the progress of their own work, we opted to use a ‘ticket’ system, which is very common in Agile workflows. Each task (or ticket) was first created as a vague ‘research’ task that required research and scoping. A completed research task would result in the creation of many new tickets that could be worked on by developers containing sufficient detail in order to achieve the task at hand as well as a notion of order and dependence between these tickets. The group member who completed the relevant research ticket also became the owner of the work contained and was encouraged to take ownership and be ready to provide clarification when something was unclear.

Our project board served to organise these tasks into logical categories while showing a clear progression in the status of the ticket. The original categories that we had stemmed from GitHub’s Kanban template and were the following:



Following our first consultation meeting with Dr Robert Chatley we decided to make fundamental changes to our board which would persist for the remainder of the project. First, we changed the categories to this progression:



and inverted the order to emphasise that it is preferable to complete 80% of tickets 100% of the way rather than 100% of the tickets only 80% of the way. On this board

work moved from the right to the left. It was not a lot of work for the group to get used to this new board as most of the tickets would move automatically thanks to GitHub’s automation feature. With work moving from the right to the left, whenever the board is first opened the tickets that need to be code-reviewed show up first. We decided that the order of work for everyone should be to 1. review all tickets in the ‘needs review’ column, 2. complete all tickets in the ‘needs development’ column, etc. This way we ensured that all completed tickets were reviewed by at least one person and that the review happened as soon as possible.

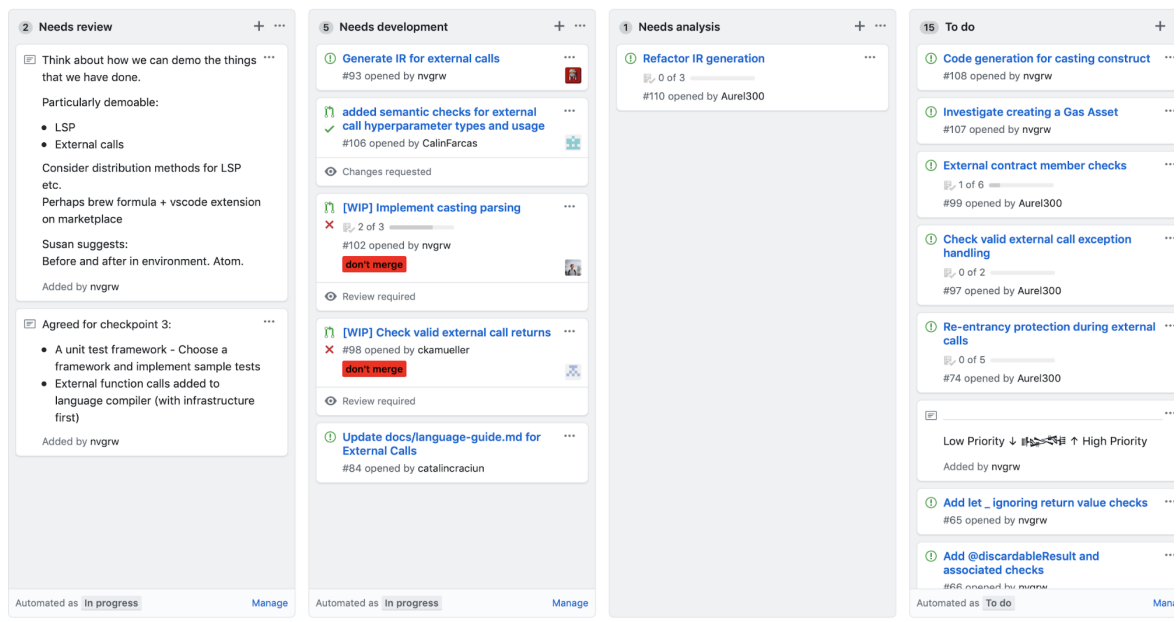


Figure 3.2: The project board at the start of the third iteration.

In addition to the board, we used GitHub’s milestones feature to track the progress on some of our ‘sub-projects’ such as external calls, unit testing, and asset traits. Milestones provide a handy progression bar that reflects the tickets’ current status on the board.

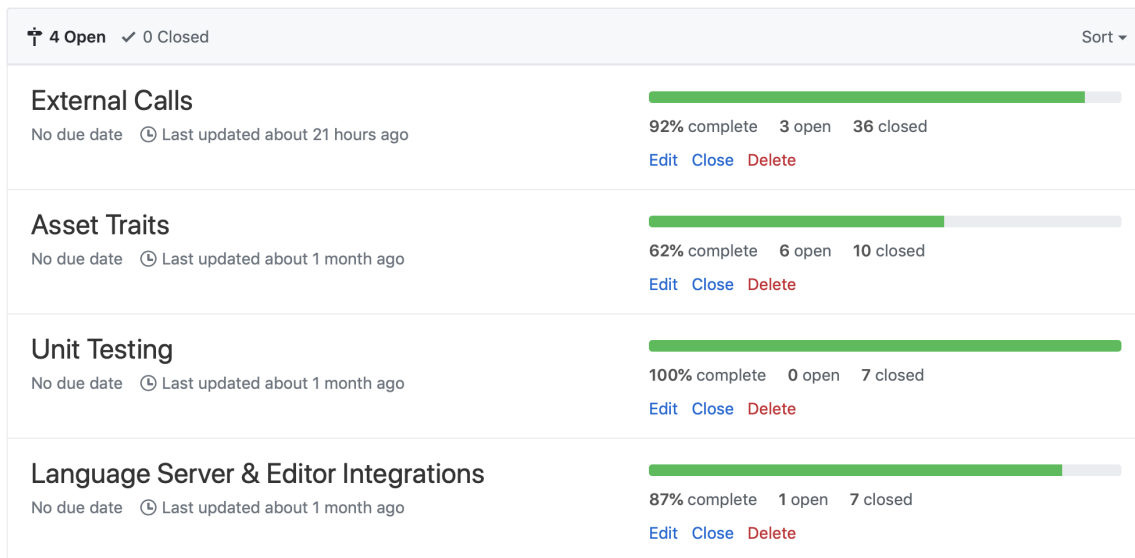


Figure 3.3: Milestones on our GitHub repository.

3.2.3 Review Process

Each ticket corresponded to a ‘feature’ branch on our GitHub repository. We made an effort to enforce a strict review process with a continuous integration pipeline which ensured that every patch worked both before and after merging, as well as to ensure that the code being committed complied with our codebase style rules. To enforce style, we used a tool called `swiftlint` which would make our build fail if there was an extraneous space or other formatting issue. Once a ticket was nearing completion, a corresponding pull request was created which referenced the issue number of the ticket. This meant that the ticket would automatically be closed when the pull request was closed due to merging or due to being superseded by another ticket or pull request.

Generally our group was very flexible about whose approval was required for any particular feature to be merged. Originally we did not want roles but it very quickly became apparent that some hierarchy was required to make progress. As a result, most code reviews would be reviewed by the ‘project master’ as well as a number of group members before merging. Occasionally when it was necessary, group members also had the freedom to bypass the review process from the project master and approve code changes themselves. This flexibility was a tradeoff between moving fast and ensuring that the code quality remains high.

















<input type="checkbox"/>	 External traits constructor ✓	#109 by ethanluoyc was merged on Nov 16 • Approved  4 of 4 <div><div></div></div>	 12
<input type="checkbox"/>	 added semantic checks for external call hyperparameter types and usage ✓	#106 by CalinFarcas was merged on Nov 21 • Approved  External Calls	 30
<input type="checkbox"/>	 External call hyper parameter checks ✓	#105 by Aurel300 was merged on Nov 15 • Approved  External Calls	 8
<input type="checkbox"/>	 Update language guide with asset traits (Checkpoints 1 – 2) ✓	#103 by Aurel300 was merged on Nov 15 • Approved  Asset Traits	 6
<input type="checkbox"/>	 Implement casting parsing ✓	#102 by nvgrw was merged 27 days ago • Approved  3 of 3 <div><div></div></div>  External Calls	 12

Figure 3.4: Some merged pull requests. Note the number of comments on the right-hand side.

We merged features only through squashing and merging in to the mainline master branch. This branch was protected from pushes and could only be modified through the GitHub user interface on pull requests to enforce the review process. A squash, in git parlance, refers to the act of taking a set of commits and combining them into a single commit. Squashing commits is a very common practice in large companies utilising source control and particularly in those that utilise a monolithic repository (monorepo) for dependency management. Additionally it meant that group members did not feel the need to measure themselves by their number of commits, as the only measurable metric was the contribution of features, which were far more important for the project. Enforcing the squashing of commits kept the history of the repository cleaner, made it easier to revert changes should they need to be reverted, as well as allowing group members to make many commits and experiment within their own branch in order to deliver a feature that was the best that it could be.

3.2.4 Communication

Our primary means of communication was Slack. Although the team was slow to adapt to Slack initially, preferring a Messenger chat instead, we soon warmed up to it and made full use of its features. In our Slack group we had some key messaging channels: #general was for communications that concerned the project in general, or for anyone needing help or clarification on something; #ask-for-code-review had a GitHub bot that would automatically ping all group members as soon as there was any activity on the repository (merges into master, opening and closing of pull requests, new issues created, etc) in order to keep all members in the loop; #standups had a daily message at 11:00 and 19:00 to remind everyone to write a brief message about what they were working on.

3.3 Task Allocation and Roles

We had originally planned to split our group into two sub-groups where each sub-group works on one aspect of the project, specifically the integration with the language server and the language extensions. We ended up primarily focusing on the language extensions and therefore this clear ‘split’ only persisted for about a week into the project. More often than not, we had a team member take charge of a particular task and delegate as needed.

We tried to encourage pairing as much as possible in order to meet our prescribed bi-weekly deadlines for the milestone assessments. If a team member struggled to complete their feature by a deadline they were constantly encouraged to get help from another team member to get their feature completed or even to hand it off to another group member.

Unlike commonly done in other Agile workflows, we chose to not allocate tasks to any individual at the task creation stage. This saved us significant time in meetings and allowed us to focus on making tickets descriptive and ensuring that everyone was on track. Instead, the group worked on the basis that everyone would put in roughly the same amount of work and could therefore pick up a ticket that was not already being worked on from the backlog once there were no more patches to review. New tickets were always spawned from research tickets and prioritised at the Monday meetings to conform with our milestone commitments to our supervisor. Despite our original idea of not having roles, at some point the group needed leadership so a leader (project master) was elected:

Member	Role
Aurel Bílý	Developer
Catalin Craciun	Developer
Calin Farcas	Developer
Yicheng Luo	Developer
Constantin Mueller	Developer
Niklas Vangerow	Developer and Project Master

Developers contributed to the project as described above, by creating feature branches, pull requests, and merging these features. The project master had the additional responsibility of organising and leading group meetings, ensuring that checkpoint documents were completed on time, as well as being the ultimate authority on code that enters the master branch. Our group was small and it was impractical to only have 5 developers, so the project master role also involved normal feature development.

Chapter 4

Design and Implementation

This section will focus on the specific design and implementation choices that we made whilst working on the Flint compiler.

4.1 Existing Codebase

We have built on top of an already existing codebase as part of our project. At the time that we began working, this consisted of a demo-able Flint compiler written in Swift, which compiled a Flint source file to a Solidity contract with YUL IR in an assembly statement representing the code of the Flint contract. The compiler processes the Solidity contract with the Solidity compiler to produce EVM bytecode. It was mainly structured as follows:

4.1.1 Lexer and Parser

A lexer which tokenized an input source file.

A handwritten, recursive-descent parser. This created an Abstract Syntax Tree (AST) from a tokenized input file, possibly outputting meaningful error messages in case of parser errors.

There were also a number of existing classes for the AST nodes; we will not go through all of them here, but they covered the necessary parts of the language, such as contracts, declarations, expressions, and components.

4.1.2 AST Passes

A number of AST passes working on different levels of the compilation process once the AST was available. These were invoked using an AST visitor, which specified the order of calling a `process` and a `postProcess` function for each AST node, with the default implementations of these doing nothing. Default stub implementations were specified in a protocol that all pass implementations had to conform to. Then, each pass only had to provide implementations for the functions relevant to its role.

Some of the existing passes were the Semantic Analyzer, which checked for semantic errors in the input, such as trying to declare the same function twice; the Type

Checker; and the IR Preprocessor, which applied final transformations to the AST right before code generation, such as copying default implementations from traits. An environment was used to collect information about the program. Each of these passes could add errors, warnings, and notes to a list of diagnostics. If any errors were encountered, the compilation would not succeed. Notably, the IR code generation itself was not implemented using the existing AST pass template, but rather a custom-made visitor pattern.

4.1.3 Example Files and Tests

The Flint project provided a number of invalid and valid source files as examples and for integration testing. The integration testing files were used to test three stages of the compilation; parsing, semantic analysis, and behaviour of the code.

The overall structure of the project in the beginning is illustrated on figure 4.1.

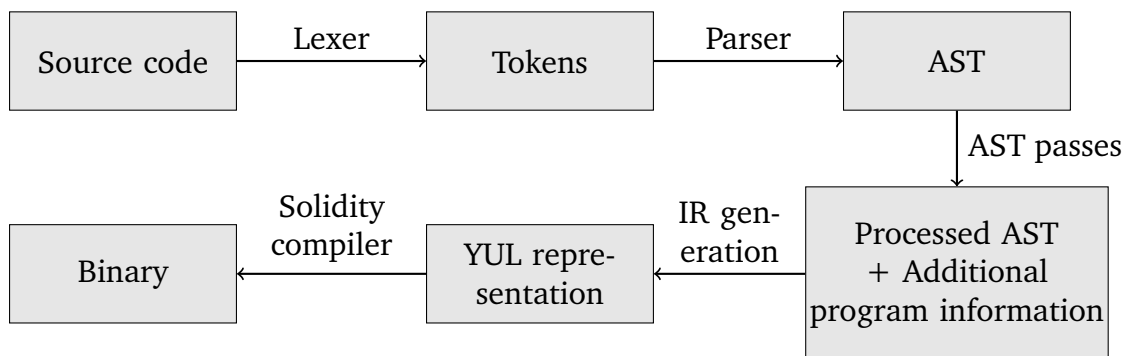


Figure 4.1: Graphic representation of the Flint compilation process.

4.2 Our Approach

Starting from the existing codebase, our aim was to maintain consistency as much as possible, while adding our own features and improving the compiler where necessary. As such, we have kept the infrastructure described above and added our own AST node classes and AST passes, as well as extending the existing passes.

Most of the features we have implemented can be grouped into two categories: improving the language and IDE integration. We have also added unit testing and enhanced the code generation.

4.3 Improving the Language

Improving the language involved implementing new features and fixing problems with the compiler.

- Asset traits – traits that capture the behaviour of assets such as Wei and express this in an extensible manner.

- #28 Add self token to parser
 - #31 Add struct trait Asset into stdlib
 - #33 Polymorphic Self for defaulted function declarations in struct traits
 - #38 Concrete Self in non-defaulted function declarations
 - #47 Adding Self to Asset Trait, fixing init and type checking issues
 - #56 Remove stdlibType from types
- External calls – calling Solidity contracts deployed on the blockchain to be a part of the network.
 - #68 Parse do ... catch statements
 - #69 Parse if let ... statements
 - #70 Parse external calls
 - #71 External traits
 - #72 Semantic checks for external calls
 - #73 Generate IR for external calls
 - #74 Re-entrancy protection during external calls
- Adding unit testing – adding necessary testing frameworks and foundational tests to better understand the compiler behaviour.
 - #75 Set up testing
 - #86 Unit test part of SemanticAnalyzer
 - work on updating Cuckoo and its dependencies
- Refactoring code generation – improving the extensibility of the code generator.
 - #127, Add definition for YUL types
 - #110, #127 Refactor IR generation
 - #127 Use the refactored codegen to handle try-catch blocks
- Updating the documentation
 - #139 Updated language guide

We have also improved existing features such as the associativity of binary expressions after parsing and the declaration syntax of events.

In total we made changes to 391 files, with 12,153 additions and 2,157 deletions.¹

¹<https://github.com/flintlang/flint/compare/master...flintrocks:master>

4.3.1 Asset Trait

One of our first goals was to add an Asset trait to the Flint standard library. The purpose of this was to enable assets other than Wei to be declared, while still having common shared functionality without duplicating the same code. Less duplication means that it is less likely for a user to forget to re-implement the safety features of the transfer method, as well as only allowing transfers to be made between two assets of the same type.

The problem with this was that the Asset trait would need to use a generic type for some of its method signatures, such as the transfer method. This is because using Asset would not ensure that transfers only happen between the same asset types, as Asset theoretically represents all possible asset types. Flint did not have generics.

We identified two solutions to the problem.

The first one was to implement generics with type constraints in Flint. This requires support for generics with type constraints. The asset trait declaration involved includes bounded type parameters in the form `struct trait Asset<T: Asset<T>>`, similar to how Comparable is declared in Java². An asset such as Wei could be declared as `struct Wei: Asset<Wei>`, and this way the concrete type Wei would be recorded in the declaration and used for the trait methods.

The second solution was to implement a polymorphic Self type. Any occurrences of Self in a trait signature would be replaced with the concrete type of the asset structure in question, preventing the programmer from using two assets at the same time as these functions would be undefined.

We opted for the second option, as it was simpler and easier to implement. From the start, Flint was designed with simplicity and robustness in mind and we wanted to honour this design principle. Adding generics to the language would have been a significant long-term undertaking requiring careful design and implementation. Additionally, generics seemed like a feature which could potentially introduce vulnerabilities into contract code due to bugs in the way that functions are called, and safety is one of the most important tenets of the Flint language design. We therefore deemed it preferable to keep the language simpler for the sake of limiting the impact of bugs introduced into the compiler.

For the purposes of illustration, here is the resulting Asset trait Flint code:

```
// Any currency should implement this trait to be able to use the currency
// fully. The default implementations should be left intact, only
// 'getRawValue' and 'setRawValue' need to be implemented.

struct trait Asset {
    // Initialises the asset "unsafely", i.e. from 'amount' given as an integer.
    init(unsafeRawValue: Int)

    // Initialises the asset by transferring 'amount' from an existing asset.
    // Should check if 'source' has sufficient funds, and cause a fatal error
    // if not.
```

²<https://docs.oracle.com/javase/tutorial/java/generics/bounded.html>

```

init(source: inout Self, amount: Int)

// Initialises the asset by transferring all funds from 'source'.
// 'source' should be left empty.
init(source: inout Self)

// Moves 'amount' from 'source' into 'this' asset.
mutating func transfer(source: inout Self, amount: Int) {
    if source.getRawValue() < amount {
        fatalError()
    }

    source.setRawValue(value: source.getRawValue() - amount)
    setRawValue(value: getRawValue() + amount)
}

mutating func transfer(source: inout Self) {
    transfer(source: &source, amount: source.getRawValue())
}

// Returns the funds contained in this asset, as an integer.
mutating func setRawValue(value: Int) -> Int

// Returns the funds contained in this asset, as an integer.
func getRawValue() -> Int
}

```

Note the `Self` in the constructor methods and the `transfer` method. During semantic checking, `Self` is replaced with the containing structure type for semantic checking purposes, allowing the existing implementation to work with no modification to how traits are checked, including overriding of default implementations and trait conformance. Following the semantic analysis step, structures conforming to traits are expanded. The expansion step is performed only when a trait has a default implementation for a function and the structure does not override it. When this is the case, we create a copy of the default function implementation and replace all occurrences of `Self` with the concrete structure type, adding it to the structure body in the process, ready for code generation. Code generation generates the function body just like it would generate a normal function.

While implementing polymorphic `Self`, we encountered several problems. Initially we had assumed that trait polymorphism had already been implemented, so the idea was to implement `Self` by adding a series of checks and keeping the code generation pre-processing the same as it was before. This ended up not being enough, because trait polymorphism did not exist. We also had problems with `Self` occurring in structure initialisers since initialisers had separate argument type checks that were ignoring our replacement of `Self` with the concrete type.

Of course, this also required many additional checks to be implemented, for example

to make sure that `Self` is only used in traits and that it is replaced correctly in a variety of cases (for defaulted functions, overridden functions, etc), and also for type checking.

4.3.2 External Calls

Another important feature that we wanted to add to Flint are external calls. This is because Flint is meant to be a programming language for the Ethereum blockchain, so for it to be useful, programs and contracts written in Flint should be able to interact with pre-existing contracts written in Solidity.

We had to introduce a way of declaring these external contracts and design a syntax for calling them that would ensure safety for our programs. This is detailed in our External Calls Proposal. We summarise our conclusions below.

There were a number of problems with calling an external contract:

1. Contracts are untrustworthy by default;
2. External calls may execute arbitrary code;
3. External calls may fail silently;
4. Interfaces may be incorrectly specified.

For a language that aims to be safe, it is not hard to see why arbitrary code executed by another contract and calls that may fail silently would pose a problem. These were the identified solutions:

1. External contracts should be considered untrustworthy, and there will not be a way to change this at the time of writing.
2. External calls should always be surrounded with do-catch blocks, where any call potentially throws an error, i.e. it behaves like `try` in Swift.
3. Any data related to an external call should be specified at the call site.
4. External calls should have a syntax distinct from regular function calls.

To declare an external contract, an ‘external’ trait is used. This is similar to the contract traits that were already implemented, except for a few additional limitations due to the nature of the Solidity ABI with which we want to be compatible: type states, caller protections and default implementations can not be specified; `mutating` and `public` keywords can not be specified on functions, and `Self` can not be used. External traits also have an implicit constructor that allows an ‘instance’ to be created from an address to allow function calls, similarly to how interfaces are initialised in Solidity.

Another important issue is that Solidity has significantly more integer types than Flint, such as `int64`, `int256`, `int24`. Flint only has `Int`, which is equivalent to `int256` in Solidity. This disparity makes a one-to-one mapping of types between Solidity and

Flint impossible. To solve this, we forced external traits to be specified using only Solidity types, and introduced a casting construct to convert between Solidity types and Flint types. Currently these conversions are needed when calling an external function which has Solidity-typed parameters and when retrieving the return value of an external call to convert it back to a Flint type. This construct is effectively implemented as a runtime check which ensures that the value to be converted fits its target type by comparing the sizes of the values. Checks are omitted where they are unnecessary, e.g. when converting from a smaller integer type to a larger integer type. Types such as strings are also not checked at runtime as no conversion is necessary.

This is how an external trait is declared:

```
external trait External {  
    func test() -> int256  
    func test2(param: int48) -> int24  
}
```

Then, an external call should specify: the external trait instance, the function name and its arguments, and potentially its gas and Wei allocation. The last two are not part of the function's signature, rather they are special values given to the external function to use/consume; they are implicit in the EVM. We called them 'hyper-parameters' because they constitute parameters of the transaction rather than the transaction payload itself. `value` denotes the amount of Wei attached to the transaction to be transferred to the recipient address, and `gas` is the maximum execution cost that the external call is allowed.

All external calls must be preceded by the 'call' keyword, which differentiates them from normal function calls and allows provision of hyper-parameters. Below we illustrate the general structure of an external call:

```
call! ext.functionWithArguments(value: 1 as! int256, tax: 2 as! int128)  
// note the hyper-parameters:  
call(value: Wei(100))! ext.expensiveFunction()  
call(gas: 5000)! ext.simpleFunction()
```

In order to increase safety and make external calls more useful, we introduced 3 'modes' for invoking them. Below we assume the existence of the relevant external contracts and functions.

Default (Safe) Mode

In this mode, the external call must be located in the `do` part of a `do-catch` block. If any call inside a `do-catch` block fails for any reason such as running out of gas, then the `catch` part of the block is executed. Here is an example of this mode:

```
do {  
    call ext.simpleFunction()  
} catch is ExternalCallError {  
    // recover gracefully  
}
```

Forced (unsafe) Mode

In this mode, the `call` keyword must be followed by an exclamation mark and it does not have to be in a `do ... catch` block. If the call fails, a transaction rollback occurs. Here is how this mode is used:

```
let ext: External = External(address: 0x0000...)
let x: Int = (call! ext.test()) as! Int
```

Note the use of the external trait constructor to create an instance of `External` from the given address and the use of the casting construct (`as!`) to convert the result to a Flint integer. Casting modes are similar to call modes, but we currently only support casting with `as!`, which reverts the transaction if the type conversion fails.

Optional (safe) Mode

Another part of our external call proposal is the Optional mode. The optional mode requires us to also introduce an `Optional` type into Flint and support similar semantics to Swift. Because of time constraints, we have only implemented this in the parser and partially in the semantic analyzer, but not in the code generator.

In this mode, the `call` keyword is followed by a question mark and it does not have to be in a `do-catch` block. Instead, it has to be used in a `if let` construct, similar to `Optional` types in Swift.

```
if let x: Int = (call? ext.returnInt()) as! Int {
    // function returned a value, here available as 'x'
} else {
    // no value returned, handle gracefully
}

if let y: Bool = (call(gas: 5000)? ext.returnBool()) as! Bool {
    // function returned a value, here available as 'y'
} else {
    // no value returned, handle gracefully
}
```

Implementing external calls involved changes on all parts of the compiler. We modified the grammar, the lexer, and the parser in order to support all of the new constructs introduced.

We had to modify the AST passes, most notably the semantic checker and type checker, in order to make sure calls are used correctly: hyper-parameters are passed where needed and their types are correct, external trait declarations are valid according to the rules above, etc. Code generator changes that we had to make included generating the IR to call an external Solidity contract, casting between different types of integers, and recovering from errors in `do-catch` blocks.

4.3.3 Adding Unit Testing

After working for a few weeks, it became increasingly clear that the lack of proper unit testing was becoming very problematic, since we only had few signals of feature regression. While some tests already existed, they were rather ad-hoc and focused on the overall behaviour of the compiler; there was no unit testing or mocking to allow us to test smaller components in isolation, which would make it easier to track down the source of a problem.

Setting up testing was a major challenge as Swift is a new language. From the start we wanted to have the ability to mock and stub protocols and classes to write effective unit tests but we found that there were several problems with this: First, Swift has very limited reflection capabilities. Reflection in Swift is read-only, which means that any kind of run-time stubbing like done in JMock³ and other Java testing frameworks is impossible. Second, the ecosystem is not yet very mature. Swift frameworks are often created by independent developers as industry adoption has been fairly slow. There are several testing frameworks on the Internet but all of them suffer from lack of maintenance, often being outdated and incompatible with Swift 4.2, the version of Swift that we are targeting.

Overcoming the reflection issue meant taking an approach that is applied in other languages, such as Go, that lack powerful reflection capabilities, which is static code generation. Essentially, we create stubs and mocks for all of our protocols and classes ahead of time by analysing the source code, and compile these into our unit testing target. To enable mocks to be used in place of concrete implementations, Swift forces us to use protocols or classes as structures cannot be subclassed.

Implementing this type of code generation from scratch is a significant time investment. In our search we had attempted to integrate several different frameworks, including SwiftMock⁴ and had even considered to write the code generation or boilerplate ourselves. After many hours of searching, we found a framework called Cuckoo⁵ which met all of the requirements that we had:

Easy integration: We want the framework to be easy to integrate into the project and into an existing codebase with minimal code changes required. Low boilerplating: We wish to automate the majority (or all) of the workflow. Easy-to-use API: Writing tests should be familiar and intuitive, an API for mock specification and verification should be similar to other established frameworks in the industry.

Integrating Cuckoo proved to be a challenging task. Cuckoo was not updated for the version of Swift we were targeting, and wholly incompatible with Linux. Since our development environments were cross-platform and our continuous integration ran on Linux, we needed to keep our codebase compatible with both macOS and Linux. As a result, we made a fork of Cuckoo and its dependencies and slowly updated them to work on Swift 4.2 as well as on Linux. In the long term this means that there are additional frameworks to maintain, but the hope is that the changes that we have made can either be contributed back to the original projects or that the

³<http://jmock.org/>

⁴<https://github.com/mflint/SwiftMock>

⁵<https://github.com/Brightify/Cuckoo> and our fork <https://github.com/flintrocks/Cuckoo>

maintainers find some time and make their own modifications to allow Swift 4.2 and Linux compatibility.

Once we had set up mocking, we had to choose a unit testing framework. We chose XCTest as it is supported by Apple and integrated into both the Swift package manager, which is used to compile the project and its dependencies, and Xcode. The Flint compiler is organised into several ‘modules’ that allow us to use and re-use different parts of the compiler for a set of targets (tests, flintc, IDE integration, etc). With XCTest we were able to create test targets that correspond to these modules and contain tests that test code defined within those modules. Our Makefile allows us to compile all mocks and stubs prior to building the test targets themselves to guarantee that their interfaces are up to date.

Writing tests is a major undertaking and should ideally have been done from the start of the project. As we did not have much time to execute a major refactoring of the codebase to make testing easier, we decided to focus on a select few units and wrote several tests for these to demonstrate how a test can be written. Writing tests retroactively requires an intricate knowledge of the functionality and specification of each unit, but in the future, further tests need to be written to increase the code coverage. In some of our work following the availability of testing in the project, we opted to use a Test Driven Development (TDD) workflow to test our assumptions and specify our code well. Below is an example of a unit test, showing the environment of an `ASTPassContext` stubbed with a faux testing response.

```
// Do not emit a diagnostic when there are
// no undefined functions in a contract
func testTopLevelModule_contractHasNoUndefinedFunctions_noDiagnosticEmitted() {
    // Given
    let f = Fixture()
    let contract = buildDummyContractDeclaration()
    let passContext = buildPassContext { (environment) in
        environment.undefinedFunctions(
            in: equal(to: contract.identifier)
        ).thenReturn([])
    }
    var diagnostics: [Diagnostic] = []

    // When
    _ = f.pass.checkAllContractTraitFunctionsDefined(
        environment: passContext.environment!,
        contractDeclaration: contract,
        diagnostics: &diagnostics
    )

    // Then
    XCTAssertEqual(diagnostics.count, 0)
}
```


4.3.4 Refactoring Code Generation

We also improved the code generation phase of the Flint compiler, which enables us to implement code generation for exception handling and improves the extensibility of the code generator.

Prior to our refactor, the code generator was architected around simple string concatenation. This was useful as a first prototype, demonstrating that IR conforming to the YUL specification⁶ can be generated successfully. However, the implementation introduced complications that prohibited further extension and improvement, such as making it impossible to get a reference to the result of an expression.

With the initial implementation, it was impossible to generate code for external calls since we needed to store the success status of external calls, in addition to value returned. We attempted to refactor the code generator (see #127) to return a tuple consisting of a preamble and an expression where the preamble represented ‘setup’ code and the expression was simply an identifier pointing to the result of evaluating the code. However, this would have introduced massive duplication of code when we concatenated ‘preamble’ code from subexpressions. It was also fragile and error-prone due to the lack of types in the generated code.

Our new code generator is organised in a way that mirrors the APIs found in the code generator for LLVM IR⁷ and the compiler for the Rust language⁸. YUL code is generated by emitting to ‘blocks’, which feature proper lexical scoping. The new code generator is stateful and keeps track of the current block which the code generator is emitting to. This allows us to handle code generation for external calls easily with less work involved in backtracking in the code generation process. It also helped us avoid generating some duplicate code. Our new code generator allows for new behaviour to be implemented much more easily by utilising the type safety of Swift. In particular, we emit code to the current block for setup code statements, then we merely return YUL expression objects denoting the results of the setup code.

A challenge with using the YUL IR was the lack of jump instructions in the IR. This made code generation for do-catch statements difficult. When handling external calls with do-catch, the program needs to resume at the error handling block as soon as an error is detected at runtime, but without a jump instruction this was difficult. Consider the following pseudo-Flint program which includes nested `do ... catch` blocks and multiple external calls in the `do` block.

```
do {  
  call f  
  call g  
  do {  
    call h  
  } catch is ExternalCallError {  
    // block A  
  }  
}
```

⁶<https://solidity.readthedocs.io/en/latest/yul.html>

⁷<https://llvm.org/>

⁸<https://www.rust-lang.org/>


```
} catch is ExternalCallError {  
    // block B  
}
```

In our solution we avoided the need for jump instructions at the cost of some code duplication. The above example translates into the following fragment (psuedo-code to avoid clutter):

```
success_f = call f  
if (success_f) {  
    success_g = call g  
    if (success_g) {  
        success_h = call h  
        if (success_h) {  
        } else {  
            // block A  
        }  
    } else {  
        // block B  
    }  
} else {  
    // block B  
}
```

When we have multiple external calls in the `do` block, the error handling code from the nearest `catch` block needs to be duplicated multiple times, once for each possible ‘failure’ of an external call. This would be unnecessary if YUL contained syntax for handling `jump/goto` behaviour. We tackle this problem by keeping track of a stack of `do ... catch` blocks that we are in. This allows us to identify the `else` block to use once we encounter `call ...` statements. Consider the case where we generate code for `call f`, we first generate code for the function call of `f`, the success status of that call then gets assigned to a variable. We then generate an `if` statement where the `else` block includes a copy of the error handling code from the most recent `do ... catch` level we are in. We then resume the code generation by emitting code to the then block of the generated `if` statement.

Another software engineering group worked on improving the YUL IR and implementing a new compiler for YUL. Their implementation might introduce useful extensions to YUL. With this refined design, we can easily extend our YUL definitions and utilise these new extensions.

4.3.5 General Improvements and Technical Challenges

A common theme during this project has been the risk associated with modifying any part of the already existing codebase due to the technical debt accrued over time. Many features that we had assumed should work based on previous tickets and documentation were in fact only partially implemented or not implemented at

all. When implementing a new feature we often uncovered other problems which were all interconnected, making it very easy to get sidetracked by also solving the additional issues found.

Our way of mitigating this risk was to try and prioritise implementing features that were simpler overall, to keep the amount of surrounding codebase maintenance work minimal. Thus, some of our effort has also gone into refactoring, improving and fixing features that were already there.

One example of an improvement that we made was the unification of the event declaration syntax with the function declaration syntax. Events and functions in Flint are called in a very similar fashion, the main difference being that event calls are preceded by the `emit` keyword. Following a discussion with our supervisor and the previous maintainers of the compiler, we decided to make the syntax for declaring the two constructs similar.

As part of this change, we found serious issues with event calls. The types of event call arguments were not checked correctly and even though events could be declared with default parameter values, these were ignored during the compilation process. We did not expect to have such issues, and so, as part of a simple syntax change, we also added a new pass to account for adding the default parameter values to event and function calls. We also re-wrote the argument type-checking logic as well as enforcing callsite labels and argument order, like Swift does.

Below is an example showcasing the unified event syntax (note the use of default parameters):

```
contract A {  
    event Finished()  
    event Approve(address: Address, amount: Int = 0)  
}
```

We discovered that the dot operator was right-associative, a consequence of the default behavior of a recursive descent parser. The code on the right-hand side of the dot operator would be parsed first and then combined with the code on the left-hand side. As such, `a.b.c.d` would be parsed as `a.(b.(c.d))` which meant that accessing nested structs and struct fields such as `a.b.c.d`, was very unintuitive or completely impossible. To solve this, we introduced a new AST pass that performs left rotations on certain binary expressions to make them left-associative. Tests have been added to check that we now recover the correct associativity for the dot operator.

Apart from fixing the existing compiler, another technical challenge was the fact that we had to work in Swift, with which only two of our group members had previous experience. We tried to overcome this by pairing an experienced member with an inexperienced member to help them get up to speed at the start of the project. Two of our group members worked on Linux causing a bit of a delay at the start of the project because of the work required to get Swift and all of the project's dependencies to run. Our workflow changes at the start of the project, such as the introduction of the linter and reformatting of the codebase, helped the less experienced group members write better code.

Towards the end of our project, we realised the language guide for Flint was incoherently structured and not very well-maintained. The more recent features were

not documented and some documented features were no longer working in the language. The latter category went by unnoticed because there were no integration tests for these features, if they even worked before the beginning of our project. The new language guide is available in the appendix.

4.4 IDE Integration

Apart from improving the language, we have also focused on providing integration with IDEs. This was something new for Flint, as the only existing integration was a Vim and Atom plugin. We chose to integrate Flint with Microsoft Visual Studio Code because it implements the language server protocol, is easy to extend, and is widely used by developers. The details for this feature are listed below:

- #51, #46 Expose flint compilation API as a “Compiler as a Service”
- #42, #60 Implement a LSP server
- #61, #120 Implement LSP server to recompile on file change
- #37 Implement as Flint language extension as a VSCode plugin

4.4.1 Language Server Protocol

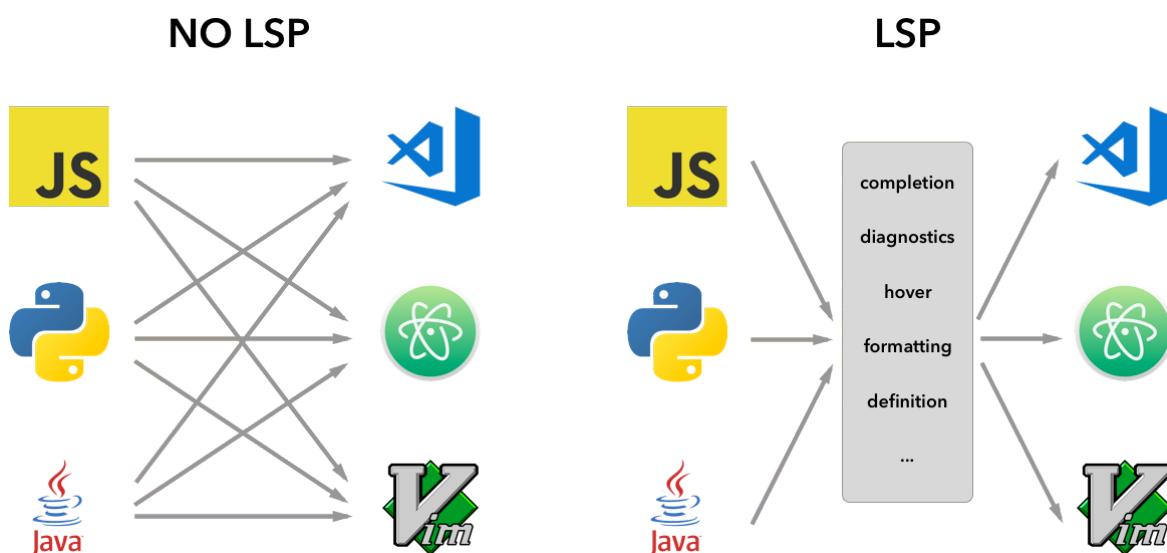


Figure 4.2: Without LSP, providing language support for Flint requires us to implement fully-featured editor support separately for all editors. With LSP, however, we can support many editors with little overhead.

LSP is a protocol which allows a single server implementation for auto-completion, go-to-definition, and diagnostics to be used for multiple editor clients. Unlike implementing a language extension for IDEs such as IntelliJ which would only benefit

IntelliJ users, implementing the LSP allows us to provide unified language support across multiple editors. Following our research, we concluded that the best way to integrate Flint with an IDE would be to implement a subset of the Language Server Protocol (LSP). We believe this is the best approach because LSP provides a standardised, documented format for the language support we aim to provide, in addition to being compatible with a range of different editors.

4.4.2 Additional Repositories

We demonstrate our IDE integration with Visual Studio Code as it features the official implementation of a language server client. To do this, we forked the repo `owensd/vscode-swift`⁹, which provides an editor extension that launches the LSP server, describe below.

We also implement a LSP server, which the editor LSP client communicates with to fetch diagnostics, etc. Since Flint is implemented in Swift, the LSP server is implemented in Swift since that avoids implementing any inter-language communication. We base our implementation of the LSP server on `theguild/swift-lsp`¹⁰, which features a complete definition of the LSP specification. The server runs as a separate target utilising the same modules as the Flint command-line interface, so we have included this in the main repository. The server calls the Flint compiler API, which we refactored to provide a ‘Compiler as a Service’, and communicate with the LSP client using TCP/IP.

4.4.3 Implemented Features

We implement a subset of the LSP server capabilities: display of diagnostics and highlighting the problematic parts of the code.

Given a Flint source file, the compiler returns a list of diagnostics, which can be errors, warnings or notes. Since information about their token positions is collected by the AST passes, the necessary information can be passed to the IDE extension. The IDE then displays all of the messages and highlights the code in question, with a different colour for each diagnostic type. This is illustrated in figure 4.3.

⁹<https://github.com/owensd/vscode-swift>

¹⁰<https://github.com/theguild/swift-lsp>

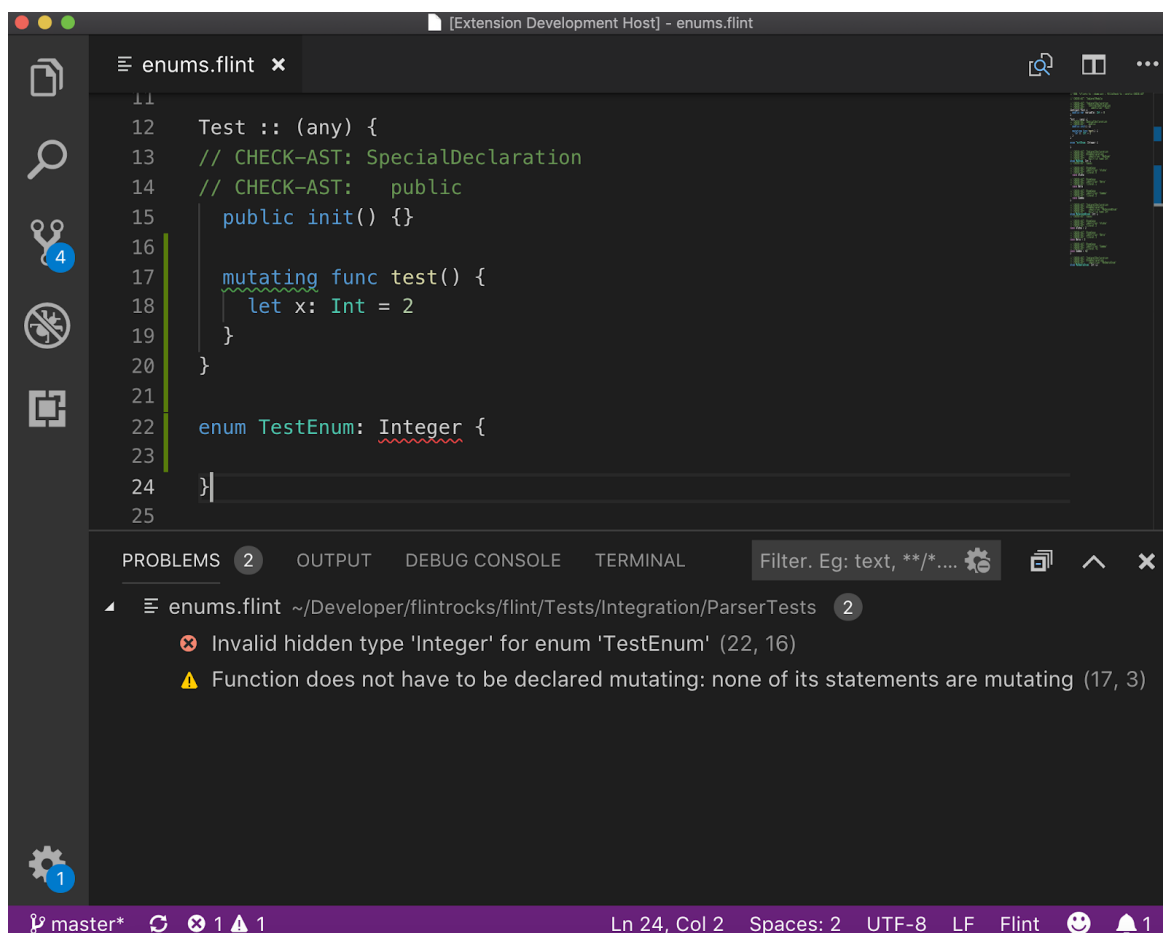


Figure 4.3: Flint LSP plug-in in use.

The second feature is automatic compilation. This works as expected: once the user stops typing for 2 seconds, the content is automatically compiled and the diagnostics are displayed.

Since every LSP message to the server is processed asynchronously, that allowed us to easily implement the timer by sleeping the thread for 2 seconds and, at the end, check if any new messages were processed in the meantime. That was possible by using global state on the LSP server, which messages can write to. After the thread wakes up from sleeping, if no new messages were processed, that means we can grab the unsaved content from the editor and save it to a temporary location, which we run the Flint compiler on. One important note is that this mechanism does not suffer from concurrency flaws with regards to the shared state and locking is not needed, since messages come through one at a time and will finish sleeping in the same fashion, so we will never have two threads waking up at the same time.

The compiler can run only on files, so saving the content to a file was the easiest option that allowed us to get this working. Another option we considered was to refactor the parser to be highly error-tolerant: the Flint compiler should be able to ingest syntactically incorrect or incomplete Flint code and still generate meaningful ASTs.

After some exploration of the current parser implementation, we believe that the

work involved in making the current parser sufficiently error-tolerant was not viable in the time given. Nevertheless, we note that there are useful articles¹¹ on implementing error tolerant parsers used in projects such as Roslyn¹² and TypeScript¹³ in the .NET ecosystem, which may be of interest to future Flint maintainers. We thus opted for the first approach as the second approach is much more complicated and does not bring any functional advantages at this stage.

¹¹<https://github.com/Microsoft/tolerant-php-parser>

¹²<https://github.com/dotnet/roslyn>

¹³<https://www.typescriptlang.org/>

Chapter 5

Evaluation

In this project we added external calls to Flint, making the language more useful for real-world Ethereum contract use. Franklin's Flint could not compile many real-world contracts because interoperating with the outside world was necessary. An example contract that might crop up in the real world can be found in the appendix. Another one of our major contributions to the compiler was improving the test suite. In particular, we added a unit testing framework, as detailed in section D. Although using this framework to its full potential by refactoring each module to be unit testable (and hence of better code quality) was out of scope of our project, this change will allow improvements of the code base by future contributors.

The other, pre-existing part of the test suite were the integration tests. Throughout our project we made sure that any changes passed all the behaviour tests on the CI server prior to merging them into the protected master branch. The integration tests verified the compiler's three key phases:

- Parser tests – verify that code was parsed into the correct AST.
- Semantic tests – verify that the proper semantic errors were emitted where expected.
- Behaviour tests – verify that the contracts produced by the compiler work correctly in an actual EVM environment (tested with the Truffle¹ suite).

Each language feature we added into Flint was also accompanied by relevant parser, semantic, and behaviour tests. These tests were usually written prior to the specific feature implementation, e.g. in the ticket description. The ticket itself was created by whoever took the original analysis ticket and any further elaboration was then expected from them. The ticket description with accompanying code fragments and tests served as a kind of specification for the new feature, emulating TDD practices. In terms of practical usability, Flint was greatly improved with the introduction of external calls, also detailed in section D. There still exist some limitations due to the intentionally simple type system in Flint, so a mapping from any Solidity type to a

¹<https://truffleframework.com/>

Flint type is not always possible. However, we have demonstrated calls to Solidity contracts in the Truffle testing environment, as well as the Ethereum Remix IDE². Our integration with the Language Server Protocol opens up Flint to new developers and make the language more approachable than ever. Receiving instantaneous feedback on code correctness is crucial to the user's understanding of the language and our integration with Visual Studio Code in particular makes writing smart contracts in Flint easier than before.

²<https://remix.ethereum.org/>

Chapter 6

Reflections and Future extensions

We are happy with what we have achieved in terms of features implemented on the existing compiler. Overall, we believe the project was a success and we hit all of the goals agreed upon with our supervisor and added further extensions to improve the quality of code and development workflow.

We think that our rigorous process helped to keep everything in order and allowed us to move towards our goals quickly. One of the most important parts of the process were the daily stand-ups on Slack: we think it played a major role in keeping every member of the team in sync. The Kanban method allowed us to keep issues flowing through the stages of development with little friction. Our Kanban board paired well with the GitHub Slack integration, ensuring that issues did not wait for a code review for too long. Some of the members of the team had no previous practical experience with Agile development beyond the Software Engineering classes and this project presented itself as an opportunity to get experience an Agile development process.

We feel that throughout this project we learnt a lot of things, both technical and non-technical. One of the most important topics we gathered a lot of insight about is the Ethereum ecosystem, which few of the team members had in-depth knowledge of when we started. The nature of the project required that we had to get familiar with all aspects of the smart contract development process including the lower-level details of the ecosystem, such as Solidity, YUL, and the EVM. Only two members of the team had previous experience with Swift, so the others had the chance to learn about Swift Programming.

At the start of the project, we decided that we will run code reviews for every merge request and rigorously kept doing so throughout the project, with some reviews becoming extremely long discussions with multiple stages of change requests. This not only helped us to ensure a high code quality standard, but we believe it also played a role in improving our own code style and learning good practices from our teammates.

We enjoyed the iterations and regularly meeting with our supervisor, which felt like working for a client, with everything that involves including meeting deadlines and the pressure of developing a good product.

6.1 Extensions

There are many parts of the compiler that can be improved and features that are desirable to have. These are just a few:

- **Modularisation** Adding modules allows a contract definition to be split across multiple source files and for the inclusion and distribution of user-made libraries and frameworks.
- **Linear types** Linear types are a more integrated version of the Asset traits that we have implemented. These types might be implemented with additional syntax and memory operations, allowing variables that hold quantities of an asset that can be split atomically, never duplicated, and never destroyed. For example, with an asset such as Wei in the context of a `@payable` contract function, this means we add a semantic check that ensures that the entire value of the incoming asset value is transferred to another instance to prevent its destruction.
- **Gas asset** A built-in Asset type, similar to Wei, that is used for the ‘gas’ hyperparameter of an external call.
- **Generics** At some point it may be desirable for the Flint language to feature generics. This feature requires careful design and consideration to stay compatible with Solidity contracts. Options include allowing generics only internally to a Flint contract, developing a Flint ABI that allows generics and is independent of Solidity, or encoding generics into the Solidity ABI itself.
- **Flint-contract interop without Solidity ABI** The Solidity ABI severely limits Flint’s capabilities to provide safer functionality when interoperating with other contracts. A Flint-specific ABI would allow us to encode additional information about types (such as type states, generics, exceptions) and introduce a trust model for external contracts.
- **Targeting eWASM¹** eWASM is a new Ethereum backend, set to replace EVM in the future. By targeting eWASM and potentially replacing the YUL backend keeps Flint up to date with current developments in the Ethereum community and makes it future-proof. With WASM being an industry standard backed by large players in the tech industry, it is poised to become widely implemented, allowing Flint to run on more platforms.
- **Flint package manager²** A package manager has long been a requested feature, but until we implemented external calls there was no reason to have one because there would’ve been no way to call another contract. Additionally, since the language does not currently support modules, there is no way to import code from another Flint program. Once modules and a Flint ABI are supported, there will be a better case for having a package manager.

¹<https://github.com/ewasm/design>

²<https://github.com/flintlang/flint/pull/151/files>

- Contract trust model Alongside the package manager, we also wanted to support a trust model between contracts. Under this model, a distinction is made between Flint contracts that have been verified and are guaranteed to work safely and contracts that have issues causing them to misbehave. The compiler might then refuse to compile code that is calling an unsafe contract without taking necessary precautions such as handling an error case.
- LiquidHaskell-style verification support³ LiquidHaskell defines logical predicates that allow the programmer to enforce properties at compile-time. In line with the safe programming aspirations of the Flint language, an extension to the compiler is to support predicate annotations that are similar in function and syntax to aid the compiler in verifying correctness and support the issuance of certificates of safeness.

We believe that with our contributions, future development will be easier and also less error-prone. With unit testing set up, we have created a solid foundation for future development. Due to time constraints, we were not able to write unit tests for every part of the compiler. Instead, we chose to adopt an incremental technique, writing unit tests along the way as we developed or changed parts of the code. In future development and as an extension to the compiler, the test suite should cover the code base fully.

Another area of improvement is the diagnostics emitted during compilation, which could benefit from more explicit and descriptive semantic analysis errors. The way that compiler passes are currently optimised is not optimal and makes unit testing difficult. A possible improvement is to split the passes based on their function even further and into separate files and units to make them more modular and easier to test. To aid with programmer reasoning, the semantic analyser pass could also be reduced to the application of a series of rules, that match properties of the AST using a domain specific language similar to XPath⁴. This allows semantic checks to be categorised in code and kept separate from other, unrelated semantic checks.

A further extension to the editor support for Flint would be to allow a wider range of editors to be used. Because we have integrated the LSP with the Flint compiler, we can support any editor that supports the LSP, including Eclipse, IntelliJ, and Atom⁵. Extending editor support is a matter of creating a small shim plugin for each editor that automatically manages the lifetime of the language server.

Automatic code completion is another desirable feature for the Flint language editor support. Adding automatic code completion to the compiler's language server is a large undertaking as we would need to create a parser that supports creating partial ASTs, as well as create and maintain more semantic information than we currently do. In order to pass this semantic information to the editor it would be necessary to refactor large parts of the existing codebase.

³<https://ucsd-progsys.github.io/liquidhaskell-blog/>

⁴<https://en.wikipedia.org/wiki/XPath>

⁵<https://langserver.org>

6.2 Ethics

Ethical issues are an important part to consider in every project and we carefully considered them throughout development.

One of the most critical issues is claiming Flint is ‘safe by design’, which requires a lot of responsibility. The main purpose of the Flint project is to build a language that is strict enough to act as a safety net for the programmer. Since contracts written in Flint handle real assets with monetary value, we cannot afford any compilation or runtime flaws that contradict the spirit and existing documentation of the language. During our development, we discovered instances of such bugs in the compiler when trying to implement new features. In order to support the safety claim, we decided we needed stringent unit testing.

The compiler is run on the user’s machine, so once distributed, the user is responsible for its usage. We do not collect any data at all, so ethical issues regarding data collection and privacy cannot apply. Another thing to note is the fact that the compiler is open source, so it is not just a ‘black box’. Everyone can audit the code and compile their own version of the project, having total control.

Other ethical issues that we considered are those that do not directly relate to Flint as a programming language, but the entire Ethereum ecosystem. Important ethical issues concerning the ecosystem more generally include the environmental impact of the blockchain and the anonymity of deployed contracts.

Blockchain networks have a significant environmental impact and with Ethereum⁶ being one of them, it is no exception. Greener alternatives are being developed, although few are as successful as Bitcoin or Ethereum. Unlike Bitcoin, the Ethereum foundation has a plan to reduce the environmental impact of its mining algorithm⁷. Importantly, Flint is not necessarily tied to the Ethereum ecosystem and could be retargeted in the future to work on other, greener alternative platforms.

Anonymity of deployed contracts is also a significant issue, since everything is public on the blockchain. That does not play well with certain domains like the health and financial sectors, which have very high privacy requirements. Solutions⁸ to add a layer of privacy to cryptocurrencies currently exist and significant work is being put into this area. One of these solutions was brought in by ZCash, Ethereum bringing support for one of their features with their October 2017 update⁹. ZCash implements something called ‘Zero-Knowledge Succinct Non-Interactive Argument of Knowledge’, or zk-Snark for short, based on the concept of zero-knowledge cryptography. Zero-knowledge cryptography means that the proof of possession of some information can be verified without any interaction between the prover and the ver-

⁶https://motherboard.vice.com/en_us/article/d3zn9a/ethereum-mining-transaction-electricity-consumption
2018-12-30

⁷<https://www.coindesk.com/ethereums-big-switch-the-new-roadmap-to-proof-of-stake>
2018-12-30

⁸<https://bravenewcoin.com/insights/mobius-to-bring-anonymity-of-monero-to-ethereum>
2019-01-02

⁹<https://bravenewcoin.com/insights/mobius-to-bring-anonymity-of-monero-to-ethereum>
2019-01-02

ifier, therefore not revealing anything.¹⁰

¹⁰<https://z.cash/technology/zksnarks/> 2019-01-04

Bibliography

- [1] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint. Master's thesis, Imperial College London, June 2018. pages 1, 2