Imperial College London

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 7, 2019

Contents

Chapter 1

Executive Summary

Ethereum is an open source platform that enables developers to build and deploy decentralised applications by leveraging blockchain technology[1]. In short, the blockchain enables a decentralised trust model due to computation being intrinsically tied to the progression of time by using cryptographic hashes[2]. 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[3]. 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[4].

Flint is a type-safe language syntactically inspired by Swift, designed for enabling developers to write safe smart contracts targeting blockchain platforms[5]. 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[6]. 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.

Apart from general codebase improvements, for this project we implemented the following language features: asset traits, external calls, and improved event declaration syntax. Additionally, we improved the way that code generation is performed and added a unit testing framework with a few preliminary tests. We also added support for the Language Server Protocol, allowing editor extensions to display compiler diagnostics in-line with Flint code.

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'[6]; 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
- Provide editor support to 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.

[1] `https://github.com/realm/SwiftLint`

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 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.

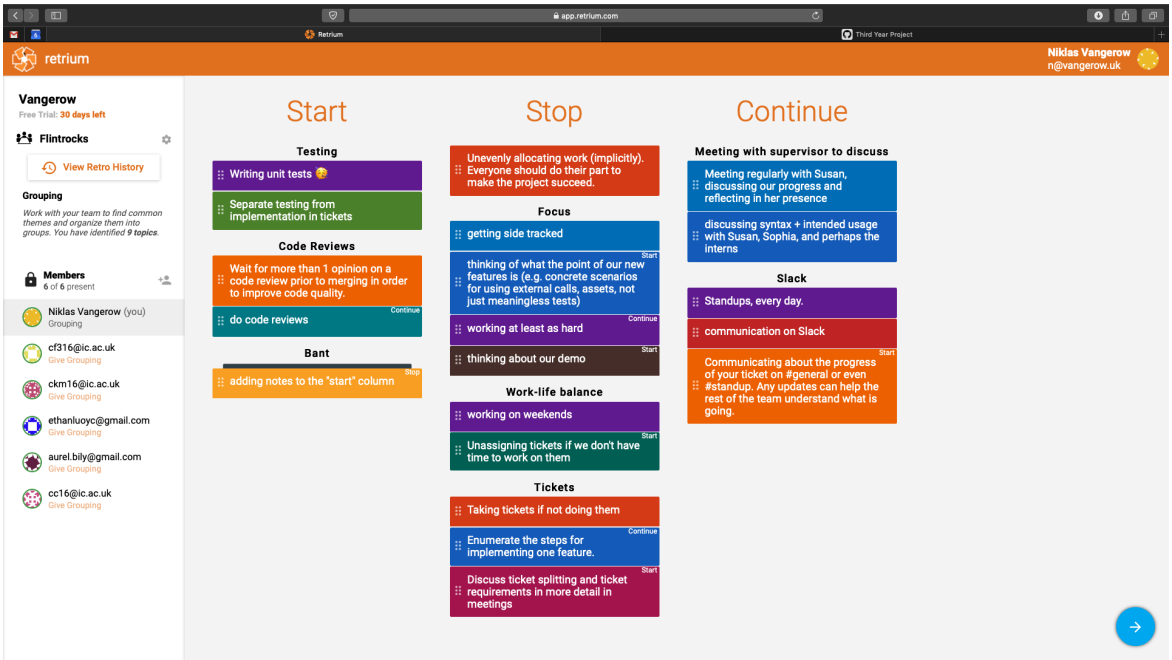


Figure 3.1: Our Retrium board showing all suggestions made by the team.

3.2 Organisation

3.2.1 Process

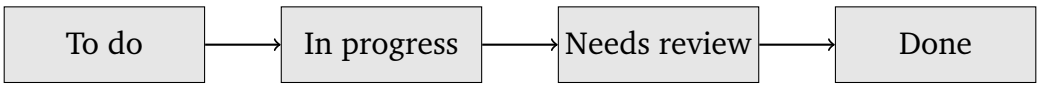
We chose Kanban as we wanted the flexibility of taking up new work as it happened while at the same time being aware of the work that is still pending. Our version of Kanban removed the WIP limit and instead enforced a softer limit that any one person could only be working on one ticket at any time. 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 shows the reconfigured board.

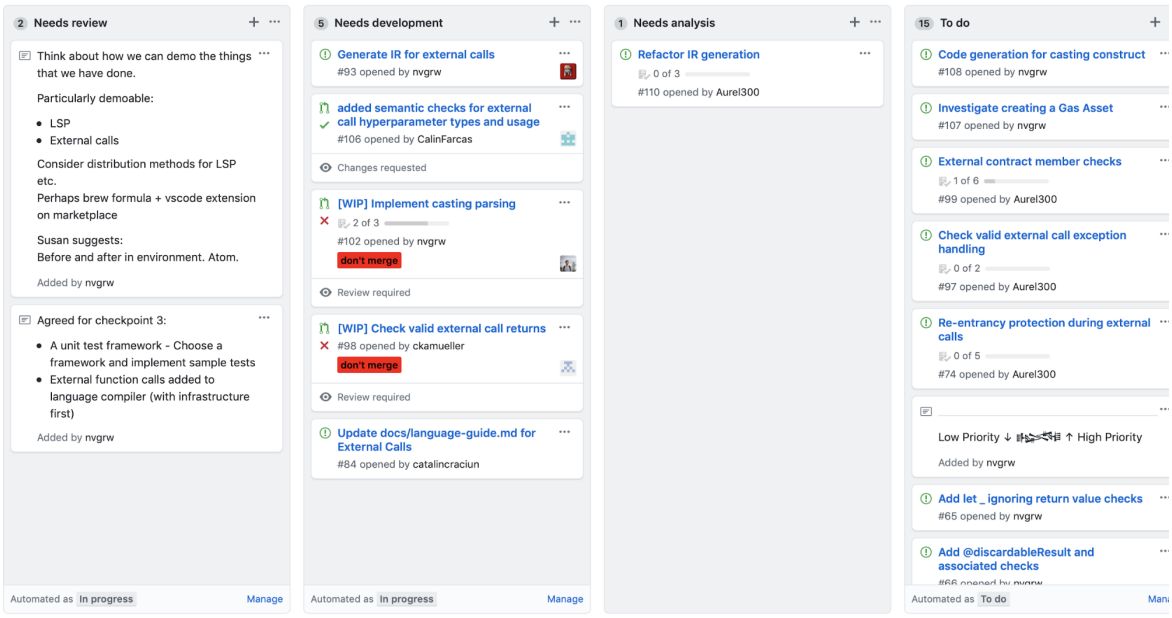


Figure 3.2: The project board at the start of the third iteration.

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. Figure 3.3 shows that milestones provide a handy progression bar that reflects the tickets' current status on the board.

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.

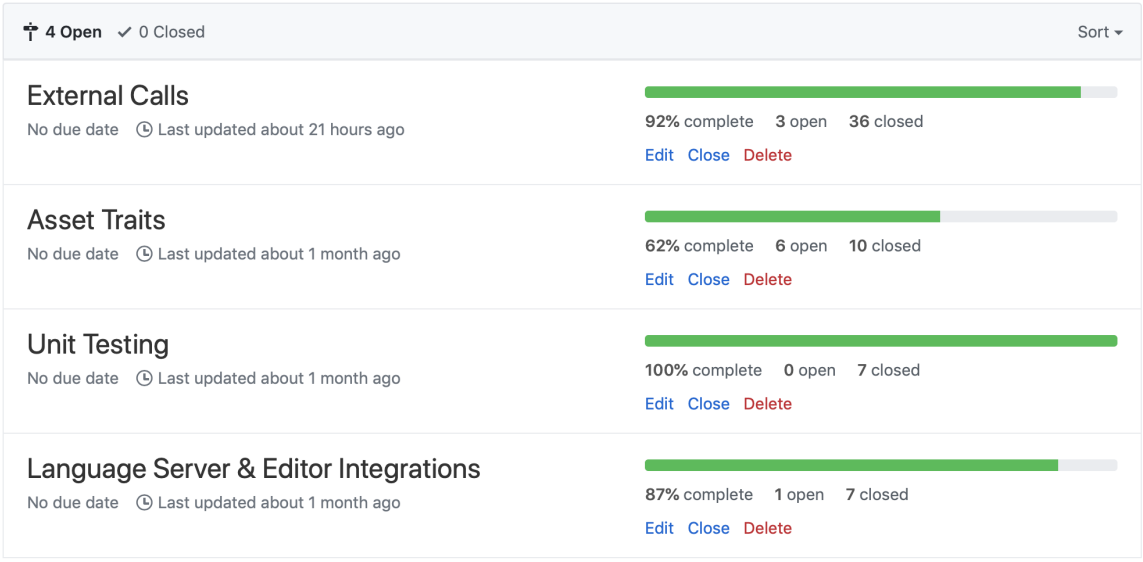


Figure 3.3: Milestones on our GitHub repository.

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 (see figure 3.4). 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.

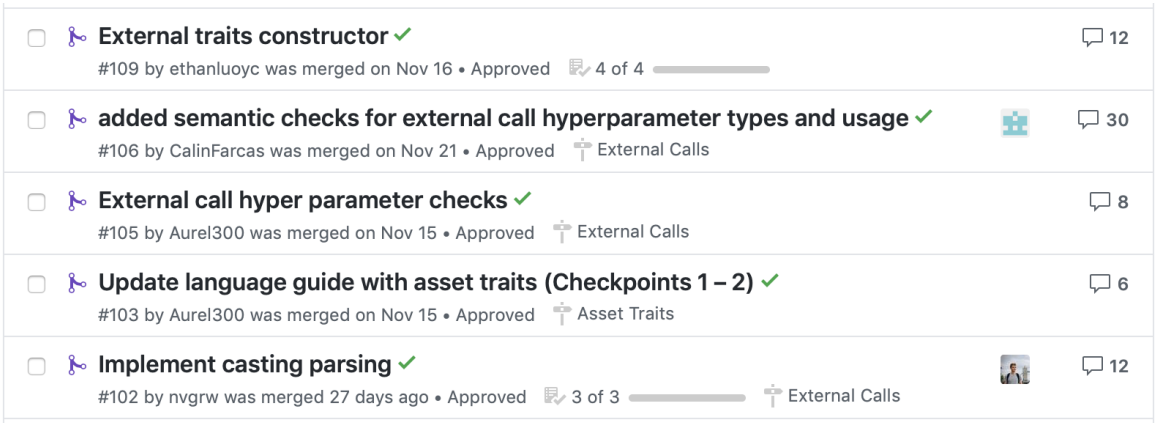


Figure 3.4: Some merged pull requests. Note the number of comments on the right-hand side.

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 subgroup 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 compilation order in the beginning is illustrated on figure 4.1.

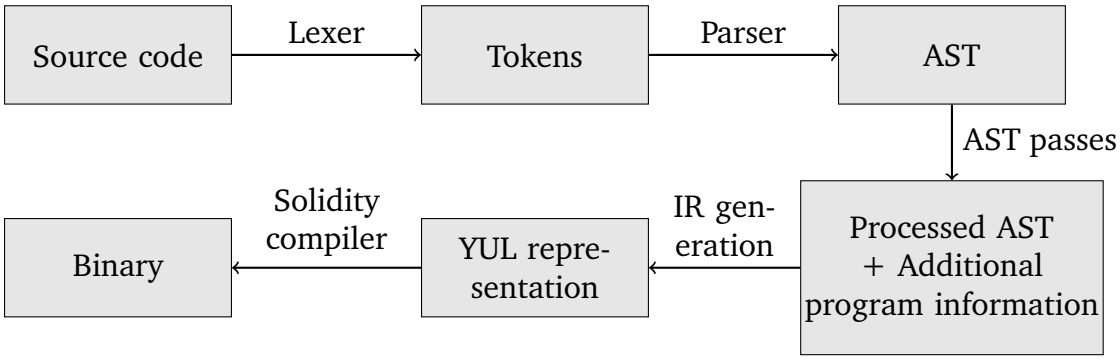


Figure 4.1: Graphic representation of the Flint compilation process.

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 Java[2]. 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 ex-

ample 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 JMock[3] 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 SwiftMock[4] and had even considered to write the code generation or boilerplate ourselves. After many hours of searching, we found a framework called Cuckoo[5] 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

[3] `http://jmock.org/`

[4] `https://github.com/mflint/SwiftMock`

[5] `https://github.com/Brightify/Cuckoo` and our fork `https://github.com/flintrocks/Cuckoo`

the 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 specification[6] 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 IR[7] and the compiler for the Rust language[8]. 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

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 (see figure 4.2). 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

[9] `https://github.com/owensd/vscode-swift`
[10] `https://github.com/theguild/swift-lsp`

Figure 4.2: Left: Significant overhead without LSP. Right: Many editors with little overhead (Visual Studio Code Contributors [7]).

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.

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.

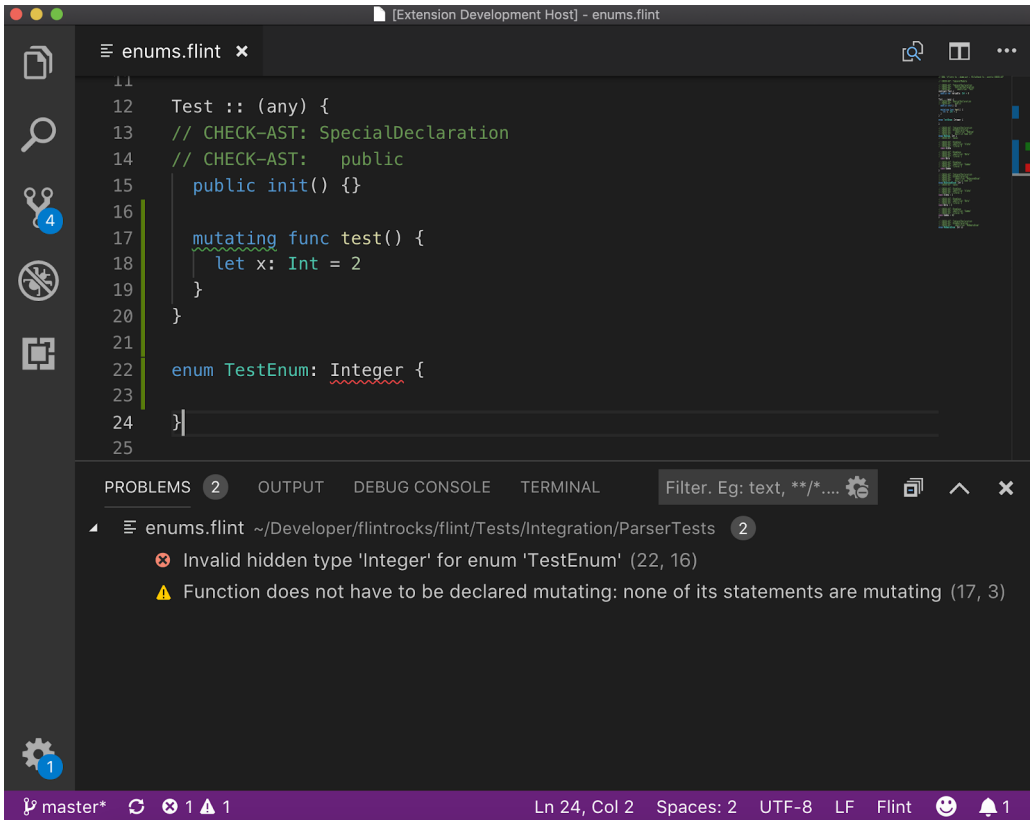


Figure 4.3: Flint LSP plug-in in use.

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 articles[11] on implementing error tolerant parsers used in projects such as Roslyn[12] and TypeScript[13] 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 chapter 4. 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 Truffle[1] 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 chapter 4. There still exist some limitations due to the intentionally simple type system in Flint, so a mapping from any Solidity type to

[1] `https://truffleframework.com/`

a 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 IDE[2].

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[5]. 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' hyper-parameter 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 eWASM[1]
  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.

[1] `https://github.com/ewasm/design`

- Flint package manager[2]
  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.

- 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 support[3]
  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 XPath[4]. 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 Atom[5].

[2] https://github.com/flintlang/flint/pull/151/files
[3] https://ucsd-progsys.github.io/liquidhaskell-blog/
[4] https://en.wikipedia.org/wiki/XPath
[5] https://langserver.org

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.

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 Ethereum[8] 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 algorithm[9]. 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. Solutions[10] 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 update[10]. 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 verifier, therefore not revealing anything[11].

Bibliography

[1] *What is Ethereum? The Most Comprehensive Guide Ever!* Sept. 2018. URL: https://blockgeeks.com/guides/ethereum/ (visited on 12/29/2018).

[2] Steve Marx and Todd Proebsting. *How Ethereum Transactions Work*. Dec. 2017. URL: https://programtheblockchain.com/posts/2017/12/29/how-ethereum-transactions-work/ (visited on 12/29/2018).

[3] Mercury Protocol. *How To: Deploy Smart Contracts Onto The Ethereum Blockchain*. Dec. 2017. URL: https://medium.com/mercuryprotocol/dev-highlights-of-this-week-cb33e58c745f (visited on 12/29/2018).

[4] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* (Dec. 2018). URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[5] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. "Writing Safe Smart Contracts in Flint". In: *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming*. Programming'18 Companion. ACM, 2018, pp. 218–219. ISBN: 978-1-4503-5513-1. DOI: 10.1145/3191697.3213790. URL: http://doi.acm.org/10.1145/3191697.3213790.

[6] Franklin Schrans. "Writing Safe Smart Contracts in Flint". MA thesis. Imperial College London, June 2018.

[7] Visual Studio Code Contributors. Dec. 2018. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on 01/05/2019).

[8] Christopher Malmo. *Ethereum Is Already Using a Small Country's Worth of Electricity*. June 2017. URL: https://motherboard.vice.com/en_us/article/d3zn9a/ethereum-mining-transaction-electricity-consumption-bitcoin (visited on 12/30/2018).

[9] Alyssa Hertig. *Ethereum's Big Switch: The New Roadmap to Proof-of-Stake*. May 2017. URL: https://www.coindesk.com/ethereums-big-switch-the-new-roadmap-to-proof-of-stake (visited on 12/30/2018).

[10] Kieran Smith. *Mobius to bring anonymity of Monero to Ethereum*. Aug. 2018. URL: https://bravenewcoin.com/insights/mobius-to-bring-anonymity-of-monero-to-ethereum (visited on 01/02/2019).

[11] *What are zk-SNARKs?* URL: https://z.cash/technology/zksnarks/ (visited on 01/04/2019).

Appendices

Annex A

External Calls Proposal

A.1 Introduction

Contracts can be created 'from outside' via Ethereum transactions or from within Flint Contracts. They contain persistent data in state variables and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM Function call and thus switch the context such that state variables in the old context are inaccessible.

A.2 Motivations

Calls to untrusted contracts can introduce several unexpected risks and errors.

When the internal contract calls to an external contract (i.e. the callee contract) the callee may execute, potentially malicious, but always arbitrary code. This code can itself include external calls to any other contract, which themselves allow arbitrary code execution and so on.

As such, **every** external call should be treated as a security risk because it requires the integrity of every contract in this chain.

However external calls are necessary to accomplish key features of smart contracts, including:

- Paying other users
- Interacting with other Contracts e.g. Tokens or Wallets

There have been several published best practice guidelines for programming with external calls (Consensys Recommendations, OpenZeppelin, Solium Security, Mythril, Solcheck). This proposal will attempt to integrate best practices into the language design. Below are causes for concern with external calls:

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.

A.2.1 1. Contracts are untrustworthy by default

```
// Bad style:
Bank.withdraw(100); // Unclear whether trusted or untrusted

function makeWithdrawal(uint amount) {
  // It isn't clear that this function is potentially unsafe
  Bank.withdraw(amount);
}

// Better style:
UntrustedBank.withdraw(100); // Untrusted external call
TrustedBank.withdraw(100);
  // External but trusted bank contract maintained by XYZ Corp

function makeUntrustedWithdrawal(uint amount) { // Name is explicit
  UntrustedBank.withdraw(amount);
}
```

It is possible to indicate trustworthiness of contracts using proper naming of functions and variables. However, this is at the discretion of the programmer and can be easily overlooked when dealing with a more complex codebase.

Instead, the language syntax itself (or the compiler) should make it plain that external calls are potentially dangerous.

A.2.2 2. External calls may execute arbitrary code

Calling functions of an external contract is also problematic since the control flow is completely taken over by the called contract and there is no way to limit exactly what the external contract will do. Consider a simple bank-like contract which stores the balances of clients (in the `put` function) and then allows clients to take out their balance (in the `get` function):

```
contract HoneyPot {
  var balances: [Address: Int] = [:]
}

HoneyPot :: caller <- (any) {
  @payable
  public init() {
    put()
  }
```

```
  @payable
  public mutating func put(implicit value: Wei) {
    balances[caller] = value.getRawValue()
  }

  public mutating func get() {
    // START OF SOLIDITY SYNTAX
    if (!caller.collectMoney.value(balances[caller])()) {
      throw;
    }
    // END OF SOLIDITY SYNTAX
    balances[caller] = 0
  }
}
```

In `put`, the contract sets the value of the `caller` address balance to zero only after checking if sending Ether to `caller` goes through.

But consider a malicious `AttackContract` designed to take advantage of the badly-written `get` function:

```
contract AttackContract {
  public init() {
    let honeyPot: Address = 0x.... // address of the deployed HoneyPot
    // START OF SOLIDITY SYNTAX
    honeyPot.put.call.value(1000)(); // so balances[caller] is 1000
    honeyPot.get.call(); // start the attack
    // END OF SOLIDITY SYNTAX
  }

  @payable
  public mutating func collectMoney(implicit value: Wei) {
    // START OF SOLIDITY SYNTAX
    honeyPot.get.call(); // collect more money!
    // END OF SOLIDITY SYNTAX
  }
}
```

The call sequence might look something like:

- `AttackContract.init()`
- `HoneyPot.put()`
- `HoneyPot.get()`
- `AttackContract.collectMoney()`

- `HoneyPot.get()`
- `AttackContract.collectMoney()`
- ...

That is, the `collectMoney` function calls `HoneyPot.get()` *before* the call to `collectMoney` is finished. This means that `balance[caller]` is never set to zero and more and more money (in multiples of the original `balance[caller]`) is transferred from `HoneyPot` to `AttackContract`.

This illustrates how easily control flow can be hijacked due to external calls.

In Solidity, `someAdress.send()` and `someAdress.transfer()` are considered safe against re-entrancy due to a workaround: while these methods still trigger code execution, the called contract is only given a stipend of 2300 gas which is currently only enough to log an event. This:

- Prevents re-entrancy attacks but is incompatible with any contract whose `fallback` function requires 2300 gas or more.
- Sometimes the programmer won't want this, but then has to fall back onto the dangerous raw calls.

In most cases, re-entrancy is not desirable, so Flint should prevent external calls to call functions of the caller (Flint) contract.

A.2.3 3. External calls may fail silently

Solidity offers low-level call methods that work on `rawAddress`: `address.call()`, `address.callcode()`, `address.delegatecall()`, `address.send()`. These low-level methods never throw an exception so they fail silently.

The following are examples of pre-exisiting solutions for external calls in solidity.

```
// Fails silently:
someAddress.send(55);

// This is doubly dangerous, as it will
// forward all remaining gas and doesn't check for result:
someAddress.call.value(55)();

// If deposit throws an exception,
// the raw call() will only return false
// and transaction will NOT be reverted:
someAddress.call.value(100)(bytes4(sha3("deposit()")));

// Better:
if (!someAddress.send(55)) {
  // Some code to handle the failure
```

```
}

ExternalContract(someAddress).deposit.value(100);
```

Flint should force the programmer to deal with potential failures of *any* external calls, by enforcing that any external call should be wrapped in a `do-try-catch` block.

A.2.4 4. Interfaces may be incorrectly specified

Minor errors in interfaces may lead to wrong code being executed. For instance, consider the following deployed contract:

```
contract Bob {
  public func set(value: Bool) {
    // ...
  }
}
```

To call `Bob.set()`, the contract `Alice` has to specify the interface (trait) for `Bob`, but it may easily be specified incorrectly:

```
contract trait Bob {
  public func set(value: Int) // note Int instead of Bool
}

contract Alice {
  func callBob() {
    let bob: Bob = 0x...
    bob.set(1)
  }
}
```

The two will produce different method IDs. As a result, `Alice` will call the fallback function of `Bob` rather than `set`, most likely with unwanted results.

This type of error is responsible for the bug in King of the Ether (line numbers: 100, 107, 120, 161)

A.3 Proposed solution

The following solution was reworked following the discussion on October 25th, with the following goals in mind, roughly in order of decreasing importance:

1. External contracts should be considered untrustworthy, and there will not (yet) be a way to change this.
2. External calls should always be surrounded with `do-catch` blocks, where any call implies a `try`.

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.
5. The supporting syntax should feel similar to Swift (wherever possible).

A valid external call should specify the following data:

- Contract (`callee`) address
- Function name
- Function arguments
- Gas allocation
- Ether (Wei) allocation

Gas allocation and Ether allocation are special values that the external function uses / consumes, but they do not form a part of its signature; they are implicit in EVM. In the remainder of the text they will be referred to as 'hyper-parameters'.

A.3.1 Code example

The function interface of the external contract has to be specified using an 'external' trait. External traits are similar to contract traits, but have a number of limitations, due to the nature of the low-level ABI of Solidity and the fact that Flint-specific features cannot be supported on Solidity contracts:

- Type states cannot be specified
- Caller protection blocks cannot be specified
- `mutating` or `public` keywords cannot be specified on functions
- Default implementations cannot be specified
- `Self` cannot be used

Some additional caveats:

- Function arguments can be given labels, but these are for internal use only (since they do not affect the ABI signature)
- Functions can be given return types, but there is no trivial way to check if a returned value is of the required type (e.g. a `Bool true` value has the same representation as a `Int 1` in the Solidity ABI)
- External traits have an implicit constructor, so that an address can be 'cast' into the trait, allowing function calls

- Functions of external trait instances cannot be called using the regular function call syntax, but must use the 'call' keyword, which also allows hyper-parameters to be specified

```
external trait Alpha {
  func simpleFunction()
  func functionWithArguments(value: Int, tax: Int)
  func functionWithReturn() -> Int
  func functionWithBoolReturn() -> Bool

  @payable
  func expensiveFunction()
}
```

The trait can then be used in Flint code. First, to initialise it from an `Address`, we use the implicit constructor of external contracts:

```
let someAddress: Address = 0x... // deployed Alpha contract
let alpha: Alpha = Alpha(adress: someAddress)
```

Then we can call functions on `alpha` using the `call` keyword, which is modeled to resemble the semantics of `try` in Swift. It has the following grammar:

```
externalCall =
  "call" WSP
  [ "(" [expression] *( "," WSP expression ) ")" ] WSP
  [ "!" / "?" ] SP
  functionCall
```

In other words, following the `call` keyword, hyper-parameters may optionally be specified, then `!` (exit on error) or `?` (return an `Optional`) may optionally change the `call` mode, then the actual external call is specificed.

Examples of (syntactically) valid uses of the `!` mode, which will cause a transaction rollback on any error:

```
call! alpha.simpleFunction()
call! alpha.functionWithArguments(value: 1, tax: 2)
call(value: Wei(100))! alpha.expensiveFunction()
call(gas: 5000)! alpha.simpleFunction()
```

Examples of (syntactically) valid uses of the default mode, which must be used in a `do-catch` block:

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

```
}

do {
  call(value: Wei(100)) alpha.expensiveFunction()
  call(gas: 5000) alpha.simpleFunction()
} catch ExternalCallError {
  // recover gracefully from either (!) failure
}
```

Examples of (syntactically) valid uses of the `?` mode, which returns an optional, and is therefore best used in a `if let ...` condition:

```
if let returnedValue: Int = call? alpha.functionWithReturn() {
  // function returned value, here available as `returnedValue`
} else {
  // no value returned, handle gracefully
}

if let example: Bool = call(gas: 5000)? alpha.functionWithBoolReturn() {
  // function returned value, here available as `example`
} else {
  // no value returned, handle gracefully
}
```

Examples of invalid uses:

```
// error: user must specify an amount of Wei to pay (@payable)
call! alpha.expensiveFunction()

// error: must be used in `if let`
call? alpha.functionWithReturn()

// error: function doesn't have a return type
if let example: Int = call? alpha.simpleFunction() {
  // ...
}

// error: return type doesn't match expected type
if let example: Int = call? alpha.functionWithBoolReturn() {
  // ...
}

// error: must use `call` for external calls
alpha.simpleFunction()
```

A.3.2 Hyper parameters

The `call` keyword accepts the following parameters:

- `gas` - an `Int` value, specifying the computational time allowed for the external call; default: `2300`
- `value` - a `Wei` value that is paid into the external contract; must be specified for functions marked `@payable`, otherwise invalid
- `reentrant` - a `Bool` value that specifies if it should be possible to call functions of the current (Flint) contract from the external contract *during* an external call (see re-entrancy problem discussed in motivation and re-entrancy discussion below); default: `false`

A.3.3 `reentrant`

Just before an external call, the Flint contract is moved into a special type state. This type state is generated automatically by the compiler, and it disallows any function to be called, preventing re-entrancy issues. After the external call is finished (no matter what the result was) the contract is placed back into the previous type state.

This behaviour may be overridden if the user chooses to do so by specifying`reentrant: true` as a hyper-parameter to the`call` keyword.

A.3.4 Implementation requirements

In the parser:

- `call` keyword, grammar for `externalCall` expression (statement?)
- `do-catch` blocks
- `if let` blocks

In the semantic analyser:

- check that `@payable` functions are given `wei`
- check that non-`@payable` functions are not given `wei`
- check that `if let ... = call? ...` calls a function with a return type
- check that `if let ... = call? ...` calls a function with the correct return type
- check that `call? ...` is used in `if let ...` (may be a parser check)
- put bound return variable in scope of `if let ...` block

In the IR generator:

- better exception handling (stack of exception handlers / addresses for each type of exception, for now only `ExternalCallError`)
- rollback on unhandled exceptions / `!` mode
- bind optional value to a variable in `if let ...`
- add special external call type state, enter into it before a call, leave it after a call

Test suite:

- add tests

A.3.5 Solidity ABI

Behind the scenes all of these interfaces are decoded into ABI function calls. ABI Specification

```
function:    sam(bytes, bool, uint[])
called with: "dave", true, [1,2,3]

0:    a5643bf2
      ^-- method ID
4:    0000000000000000000000000000000000000000000000000000000000000060
      ^-- arg1 offset
32:   0000000000000000000000000000000000000000000000000000000000000001
      ^-- true
64:   00000000000000000000000000000000000000000000000000000000000000a0
      ^-- offset 2
96:   0000000000000000000000000000000000000000000000000000000000000004
      ^-- length of arg1
128:  6461766500000000000000000000000000000000000000000000000000000000
      ^-- "dave"
160:  0000000000000000000000000000000000000000000000000000000000000003
      ^-- length of arg2
192:  0000000000000000000000000000000000000000000000000000000000000001
      ^-- arg2
224:  0000000000000000000000000000000000000000000000000000000000000002
      ^-- arg2
256:  0000000000000000000000000000000000000000000000000000000000000003
      ^-- arg2
```

A.3.6 Warnings

If the contract storage is changed after an external call (i.e. the external call modified the state) then a warning should be emitted. This should encourage two things:

1. `checks-effects-interactions` pattern. 2. `Pull over push` for external calls. This is considered a best practice as it helps isolate each external call into its own transaction that can be initiated by the recipient of the call.

```
// SOLIDITY SYNTAX

// Without push-pull
function bid() payable {
  if (highestBidder != 0) {
    highestBidder.transfer(highestBid);
    // if this call consistently fails, no one else can bid
  }
  highestBidder = msg.sender;
  highestBid = msg.value;
}

// With push-pull
mapping(address => uint) refunds;

function bid() payable external {
  require(msg.value >= highestBid);
  if (highestBidder != 0) {
    // Push: record the refund that this user can claim
    refunds[highestBidder] += highestBid;
    // Could also emit an event as an Asynchronous trigger
    // for the previous bidder to withdrawRefund
  }
  highestBidder = msg.sender;
  highestBid = msg.value;
}

function withdrawRefund() external {
  uint refund = refunds[msg.sender];
  refunds[msg.sender] = 0;
  msg.sender.transfer(refund);
}
```

A.4 Alternatives considered

A.4.1 Blind Calls

This acts as a direct translation to the ABI that gets called behind the scenes. This gives a low-level interface to the contract but is also highly prone to error.

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
func callFoo(contractAddress: Address) {
```

```
  contractAddress.call(bytes4(sha3("foo(uint256)")), a)
}
```

A.4.2 Variable binding

Results could also be bound to variables instead of an identifier:

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
let boundReturn: Int = try! alpha!.withdraw()

try let boundReturn: Int = alpha!.withdraw then {
  // Use bound return
}
```

A.4.3 Guard-like syntax

We could flip the catching of the call so you only specify the catch statement after it then continue code execution as normal. This would reduce the indentation of the language, but would then not match the if statement syntax.

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
let alpha: Director<Alpha> = 0x000... with Alpha

try alpha!.doesNothing() else {
  // If it fails
}
// If it succeeds execution will continue

try! boundReturn <- alpha!.withdraw()
// Optionally does something with boundReturn
```

A.4.4 Parameters of the call are appended

Calls need information such as the amount of gas to allocate or the Ether value to transfer. This contradicts the return type as: `address.foo` is of type `Void` and so must `address.foo.value(10)` be but `.value()` is not a property of the `Void` type. This means special cases would be needed for the type checker, and is just generally confusing.

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
contract A {
  @payable
  public func foo(i: Int){
    // ...
  }
}
```

```
func callData(address: Address) {
  address.foo.value(10).gas(800)(5)
}
```

A.4.5 Call Specification

This was rejected because it confuses both the function name, the arguments and the hyper parameters for the call. They are all assigned together.

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
call Name {
  let name: String = "foo"
  let arg1: Int = 5
  let value: Wei = 10
  let gas: Gas = 800
}
```

A.4.6 Previous version of this proposal

The following was the previous version of this proposal. Several issues have since been addressed, namely that contracts are always untrusted, that hyper-parameters were specified on a stateful `Director` (leading to potential problems when the state is specified far from an actual call), that the syntax seemed too different from Swift.

```
// THIS SYNTAX WILL NOT BE SUPPORTED!

// external contract Alpha
contract trait Alpha(State1, State2) {
  var owner: Address
}

Alpha @(State1) :: (any) {
  func doesNothing()
}

Alpha @(any) :: (owner) {
  func doesNothingWithArgs(a: Int, b: Int, c: Int)
  func withdraw() -> Int
  func deposit(value: Int) -> Bool
}

Alpha @(any) :: (any) {
  @payable
  func expensiveFunction()
}
```

```
contract AlphaUser {
  public init() {
    // Director allows external calls and setting of hyper-parameters
    let alpha: Director<Alpha> = 0x...
    // address of a deployed Alpha contract

    try alpha!.doesNothing() then {
      // Successful Call
    } catch {
      // If it fails
    }

    try! alpha!.doesNothingWithArgs(a: x, b: y, c: z)
    // Catch can not be provided by using try!,
    // then if the call fails then transaction reverts.

    try! alpha!.withdraw()
    // This flags an error as the return value is not dealt with

    try! boundReturn <- alpha!.withdraw() then {
      // Optionally do something with boundReturn
    }

    // Setting hyper parameters
    // Asset types are atomically transferred to
    // preserve special properties.
    alpha!.transfer(Wei(200))
    alpha!.allocate(Gas(2000))
    alpha.trust() // Removes the need for a bang
    try! alpha.expensiveFunction()
  }
}
```

The following features were completely removed from this proposal, since they have been deemed too ambitious / unnecessary for the time being:

- Importing trusted contracts from the Flint Package Manager
- Importing contracts from URLs
- Deploying contracts

An example of the above features:

```
// THIS SYNTAX WILL NOT BE SUPPORTED!
```

```
// Creates a contract from the data stored in Flint Package Manager
import flint:0x... as ERCToken

// Then in a function:
ERCToken.transfer(...)

import https://flint.org/examples/contract.flint as URLContract
import FileContract
import Directory.Contract

// Then in a function:
let contract: Contract<URLContract> = deploy(URLContract)
contract.argumentName()
// Value and Gas are automatically set based upon properties
```

Annex B

Language Guide

First of all, thank you for the interest in Flint!

Even though the Ethereum platform requires smart contract programmers to ensure the correct behaviour of their program before deployment, it has not seen a language designed with safety in mind. Solidity and others do not tailor for Ethereum's unique programming model and instead, mimic existing popular languages like JavaScript, Python, or C, without providing additional safety mechanisms.

Flint changes that, as a new programming language built for easily writing safe Ethereum smart contracts. Flint is approachable to both experienced and new Ethereum developers, and presents a variety of security features. Much of the language syntax is inspired by the Swift language, making it more approachable than Solidity.

For a quick start, please have a look at the Installation section first, followed by the Example section.

B.1 Getting started

B.1.1 Installation

The first step before using the Flint compiler is to install it. The simplest way is to use Docker. Otherwise, the binary packages and building from source require `solc` to be installed first.

Docker

The Flint compiler and its dependencies can be installed using Docker:

```
$ git clone https://github.com/flintlang/flint.git
$ cd flint
$ docker build -t flint .
$ docker run -it flint
```

Installing `solc`, the Solidity compiler

A non-Docker Flint install requires the Solidity compiler `solc` to be installed. For full installation instructions, see the Solidity documentation.

Binary packages

Flint is compatible with macOS and Linux platforms, and can be installed by downloading a built binary directly. Installing `solc` is a pre-requisite for using the binary packages.

The latest releases are available on the GitHub releases page.

Building from source

The Flint compiler is written in Swift, and requires the Swift compiler to be installed. See the Swift download page for latest releases.

> For older macOS machines and some Linux distributions it may be easier to use `swiftenv`. See the `swiftenv` website for installation instructions.

Once Swift is installed, Flint can be compiled by cloning the GitHub repository and invoking `make`:

```
$ git clone https://github.com/flintlang/flint.git
$ cd flint
$ swiftenv install # only if using swiftenv
$ make
```

The built binary will be available in `.build/debug/flintc`. You can then add `flintc` to your `PATH` using:

```
$ export PATH=$PATH:`pwd`/.build/debug/flintc
```

If you are planning to contribute to the Flint project or wish to run the tests, please also install:

- Node.js
- The Truffle suite - for testing contracts and running the integration tests
- SwiftLint - to ensure there are no stylistic or formatting issues with the code

B.1.2 Example

This section demonstrates the full workflow of writing a smart contract in Flint, compiling it, testing it locally using Truffle, and deploying it to the Ethereum network.

Creating a simple contract

The first step is to create a Flint source file. Our example smart contract will be a simple counter. It will have a state – the number of 'hits'. Its current value can be displayed by calling its `getValue` function. Its value can be increased by paying some Wei to its `hit` function.

We create a file called `main.flint` and put the following code in it:

```
// This is the declaration of the contract. In this simple example, it only
// includes the one state variable, `hits`.
contract Counter {
  // `hits` will initially be `0` and it will be an integer variable (`Int`).
  var hits: Int = 0
}

// These are the functions of the contract. The `:: (any)` indicates that
// these functions can be called by anyone on the Ethereum network.
Counter :: (any) {
  // This is the constructor, called when the contract is first created. There
  // is nothing we need to do here at this point, so it is empty.
  public init() {}

  // This function returns the current counter value. It takes no arguments,
  // but returns an `Int`.
  public func getValue() -> Int {
    return hits
  }

  // This function increates the counter value by one. It only does this if
  // some Wei was paid, so the function is `@payable`. The amount of Wei paid
  // is available as the implicit `value` argument, although we do not use this
  // value here.
  @payable
  public mutating func hit(implicit value: Wei) {
    hits += 1
  }
}
```

Compiling `Counter`

We can compile `Counter` to the Solidity contract file `contracts/main.sol` using the terminal command:

```
$ flintc main.flint --emit-ir --ir-output contracts
```

Testing `Counter`

Even though `Counter` is extremely simple, we should test it before deploying it to the Ethereum network.

During early iterations, it may be useful to debug a contract directly with the Remix IDE. This is detailed in a separate section.

Writing unit tests for Solidity (and hence Flint) contracts is possible with the Truffle framework. Unfortunately, it is necessary to provide some boilerplate code for the tests to work. This consists of:

1. Create a `Migrations.sol` file in the `contracts` directory (alongside the Flint-generated `main.sol`) with the content:

   ```
   pragma solidity ^0.4.2; contract Migrations {}
   ```

2. Create a `1.js` file in a new directory called `migrations` with the content:

   ```
   var Contract = artifacts.require("./Counter.sol");
   module.exports = (deployer) => deployer.deploy(Contract);
   ```

3. Create a `truffle.js` file with the content:

   ```
   module.exports = {};
   ```

Finally, we can create a `test.js` file with the unit tests:

```
var Contract = artifacts.require("./Counter.sol");
var Interface = artifacts.require("./_InterfaceCounter.sol");
Contract.abi = Interface.abi;

contract("Counter", function(accounts) {
  it("should be possible to deploy the counter", async function() {
    await Contract.deployed();
  });
  it("should be possible to increase the counter value", async function() {
    let counter = await Contract.deployed();
    await counter.deposit({value: 10});
  });
  it("should have a zero value initially", async function() {
    let counter = await Contract.deployed();
    let value = await counter.getValue();
    assert.equal(value.valueOf(), 0);
  });
  it("should be possible to interact with the counter", async function() {
    let counter = await Contract.deployed();
```

```
    var value;

    value = await counter.getValue();
    assert.equal(value.valueOf(), 0);

    await counter.deposit({value: 10});

    value = await counter.getValue();
    assert.equal(value.valueOf(), 1);

    value = await counter.getValue();
    assert.equal(value.valueOf(), 1);

    await counter.deposit({value: 10});

    value = await counter.getValue();
    assert.equal(value.valueOf(), 2);

    await counter.deposit({value: 10});

    await counter.deposit({value: 10});

    value = await counter.getValue();
    assert.equal(value.valueOf(), 4);
  });
});
```

We can then run these unit tests using:

```
$ truffle test test.js
```

Deploying `Counter`

Finally, we can deploy our contract to the Ethereum network. Note that this may cost real money and the `Counter` contract is not terribly useful!

Since Flint produces Solidity contracts, they can be deployed by following a standard guide, such as this one.

B.1.3 IDE integration

The Flint compiler has options to integrate with VS Code, Vim, and Atom, although Vim and Atom currently only support syntax highlighting, not inline error / warning display.

VS Code

Experimental support is provided for VS Code via a LSP plugin available at `vscode-flint`. The `vscode-flint` repository needs to be opened in VS Code as a project, then run in debug mode. Assuming the Flint compiler was built and the `langsrv` executable was created, this should provide errors and diagnostics in Flint files.

Vim

Syntax highlighting can be activated in Vim by using the following command in the Flint repository:

```
$ ditto utils/vim ~/.vim
```

Atom

Syntax highlighting in Atom can be obtained by installing the `language-flint` package.

B.1.4 Compilation

Flint compiles Flint source code to YUL IR wrapped in Solidity contracts. These can then be compiled into EVM bytecode using the Solidity compiler, and deployed to the Ethereum blockchain using a standard client or the Truffle framework.

A Flint source file named `main.flint` containing a contract `Counter` can be compiled to a Solidity file using:

```
$ flintc main.flint --emit-ir
```

You can view the generated code using:

```
$ cat bin/main/Counter.sol
```

The Solidity compiler `solc` can be used to compile `.sol` files to EVM bytecode. This step can be done automatically by `flintc` (internally invoking `solc`) instead by using:

```
$ flintc main.flint --emit-bytecode
```

There are more command-line options available in `flintc`. To show a full listing, use:

```
$ flintc --help
```

B.1.5 Remix integration

Remix is an online IDE for testing Solidity smart contracts. Flint contracts can also be tested in Remix, by compiling Flint to Solidity. In this example, we are going to test the `Counter` contract from the example section

After compiling the `Counter` contract, we can obtain the Solidity program in the file `bin/main/Counter.sol`. To interact with this contract in Remix:

1. Copy the contents of `bin/main/Counter.sol` and paste the code into Remix.
2. Press the red Create button under the Run tab in the right sidebar.
3. You should now see your deployed contract below. Click on the copy button on the right of `Counter` to copy the address of the deployed contract.
4. Select `_InterfaceMyContract` from the dropdown above the Create button.
5. Paste the contract address (from step 3) into the 'Load contract from Address' field, and press the At Address button.
6. You should now see the public functions declared by your contract (`getValue`, `hit`). Red buttons indicate the functions are mutating, whereas blue indicated non-mutating. You should now be able to call the contract's functions.

B.2 Language guide

B.2.1 File structure

Flint files consist of one or more contract declarations, and optionally struct declarations, trait declarations, external contract declarations, and/or enumerations.

Comments

Comments may be used throughout the source code. Comments are started with a double slash `//` and continue to the end of that line.

B.2.2 Types

Flint is a statically-typed language with a simple type system, with basic support for subtyping through traits.

> **Planned feature**
>
> Currently, the types of all constants, variables, function arguments, etc. have to be explicitly declared. Type inference is a planned feature.

Flint is a type-safe language. A type safe language encourages clarity about the types of values your code can work with. It performs type checks when compiling code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

Basic types

Type	Description
`Int`	256-bit integer.
`Address`	160-bit Ethereum address.
`Bool`	Boolean value.
`String`	String value. Currently limited to 256 bits, i.e. 32 bytes.
`Void`	Non-value. Note that the `Void` type is never directly used. It is implicit when a function has no return type.

Dynamic types

Name	Type (in code)	Description
Dynamic-size list	`[T]`	A list of elements of type `T`. Elements can be added to it or removed from it.
Fixed-size list	`T[n]`	A list containing `n` elements of type `T`. It cannot have a different number of elements than its declared capacity `n`.
Dictionary	`[K: V]`	Dynamic-size mappings from one key type `K` to a value type `V`. Each stored key of type `K` is associated with one value of type `V`.
Polymorphic self	`Self`	See polymorphic self.
Structs		Structs (structures), including user-defined structs.

Range types

Flint includes two range types, which are shortcuts for expressing ranges of values. These can only be used with `for-in` loops.

The half-open range (`a..<b`) defines a range that runs from `a` to `b`, but does not include `b`.

```
for let i: Int in (0..<5) {
  // i will be 0, 1, 2, 3, 4 on separate iteratons
}
```

The open range operator (`a...b`) defines a range that runs from `a` to `b` and does include `b`.

```
for let i: Int in (0...5) {
  // i will be 0, 1, 2, 3, 4, 5 on separate iteratons
}
```

At the moment, both `a` and `b` must be integer literals, not variables!

> **Planned feature**
>
> In the future, it will be possible to iterate up to an arbitrary value. See #397.

Solidity types

When specifying an external interface, Solidity types must be used. The types usable in Flint are:

- `int8`, `int16`, `int24`, ... `int256` (all multiples of 8 bits)
- `uint8`, `uint16`, `uint24`, ... `uint256` (all multiples of 8 bits)
- `address`
- `string`
- `bool`
- `bytes32`

See casting for more information.

B.2.3 Constants and variables

Constants and variables associate a name with a value of a particular type. The value of a constant cannot be changed once it is set, whereas a variable can be set to a different value with assignment statements.

Constants and variables of a contract are its state properties. They are data stored in the EVM storage, and even though they are not directly modifiable, they are publicly visible, so they should never hold private or sensitive data.

Otherwise, local constants and variables are declared inside functions. These are specific to a given transaction, stored in the EVM memory. Even though these terms are not part of a contract's state, if they are part of an executed transaction their values will still be recorded in the transaction history.

To declare a constant with the name `<name>` of the type `<type>` with the initial value being the result of `<expression>`:

```
let <name>: <type> = <expression>
```

The expression is evaluated once, when the declaration is executed. The expression can be complex, or just a simple literal. Examples:

```
let unity: Int = 1
let answer: Int = 7 * 6
let usingFlint: Bool = true
let digitsOfPi: [Int] = [3, 1, 4, 1, 5, 9, 2, 6]
let structExample: Rectangle = Rectangle(width: 30, height: 40)
```

If a constant is a state property of a contract, it may be given no initial value, but in that case it must be set in each initialiser of that contract:

```
let <name>: <type>
```

To declare a variable with the name `<name>` of the type `<type>` with the initial value being the result of `<expression>` (see expressions), the syntax is the same, but `var` is used instead of `let`:

```
var <name>: <type> = <expression>
```

Examples:

```
var counter: Int = 0
var areWeThereYet: Bool = false
```

The value of a variable or a constant can be used in expressions once it is declared, simply by writing its name.

B.2.4 Functions

Functions are self-contained blocks of code that perform a specific task, which are called using their identifier. They are defined with the keyword `func` followed by the identifier and the set of parameters and optional return type:

To declare a function with the name `<name>` returning a value of type `<type>`, taking the list of parameters `<parameters>`, optionally with modifiers `<modifiers>` and attributes `<attributes>`:

```
<attributes>
<modifiers> func <name>(<parameter-1>, <parameter-2>, ...) -> <type> {
  // statements
}
```

Some functions do not return a value:

```
<attributes>
<modifiers> func <name>(<parameter-1>, <parameter-2>, ...) {
  // statements
}
```

Function attributes

Attributes annotate functions as having special properties. Currently the only example of this is `@payable`. For more information, see payable.

Function modifiers

In Flint all functions are `private` by default and as such can only be accessed from within the contract body. This can be changed using access modifiers:

- `public` access enables functions to be used within their contract and exposes the function to the interface of the contract as a whole when compiled. Other contracts and users on the Ethereum network may call `public` functions directly.

- `private` access (default and not a keyword that is explicitly set) only enables functions to be used within their contract.

Examples:

```
func giveOutMoney(to: Address) {
  // only callable from other contract functions
}

public func takeMoney(from: Address) {
  // can be called by Ethereum users and contracts
}
```

Smart contracts can remain in activity for a large number of years, during which a large number of state mutations can occur. To aid with reasoning, Flint functions cannot mutate smart contracts' state by default. This helps avoid accidental state mutations when writing the code, and allows readers to easily draw their attention to the mutating functions of the smart contract.

Naturally, it is sometimes desirable to write a function that changes the state properties of its contract. This is enabled with the `mutating` modifier:

Examples:

```
contract Counter {
  var hits: Int = 0
}

Counter :: (any) {
  // This would be a compile-time error - the function needs to be declared
  // with `mutating`!
  //public func incrementA() {
  //  hits += 1
  //}

  // This can compile:
  mutating public func incrementB() {
    hits += 1
  }
}
```

Function parameters

Functions can also take parameters which can be used within the function. These must be declared in the function signature. Flint also supports parameters that take default values, but no non-defaulted parameter may follow one that has a default value.

Each parameter has the syntax:

```
<modifiers> <name>: <type modifiers> <type>
```

Currently the only possible (optional) `<modifier>` is `implicit`. See payable for more information. The only possible (optional) `<type modifier>` is `inout`. See inout for more information.

Below is a function that mutates the dictionary of peoples' names to add the key/value pair of the caller's address and the given name. If the parameter `name` is not provided to the function call, then the default value of `"John Doe"` will be used. For more information about callers, see caller bindings.

```
contract AddressBook {
  var people: [Address: String]
}

AddressBook :: caller <- (any) {
  mutating func remember(name: String = "John Doe") {
    people[caller] = name
  }
}
```

Return values

You can optionally indicate the return type of a function with the return arrow `->`, which is followed by the return type. Inside the function, a `return` statement must be used, to return a value of the same type as the declared return type.

Example:

```
func hello() -> String {
  return "Hello, world!"
}
```

If the return type is omitted, the function is considered a `Void` function, and a call to it cannot be used in expressions as a value.

Initialisers

Initialisers are special functions called to create a struct or contract instance. The syntax is slightly different:

```
<modifiers> init(<parameter-1>, <parameter-2>, ...) {
  // statements
}
```

The statements that can be used in initialisers are limited to 'simple' statements, which means no external calls, control flow statements, etc. After an initialiser is executed, all the state properties of its containing struct or contract should have a value.

Payable

(Contract-specific.)

When a user creates a transaction to call a function, they can attach Wei to send to the contract. Functions which expect Wei to be attached when called must be annotated with the `@payable` annotation, otherwise the transaction will revert when the function is called.

When adding the annotation, a parameter marked `implicit` of type `Wei` must be declared. `implicit` parameters are a mechanism to expose information from the Ethereum transaction to the developer of the smart contract, without using globally accessible variables defined by the language, such as `msg.value` in Solidity. This mechanism allows developers to name `implicit` variables themselves, and they need not remember the name of a global variable.

Functions in Flint can be marked as payable using the `@payable` attribute. The amount of Wei sent is bound to an implicit variable:

```
@payable
public func receiveMoney(implicit value: Wei) {
  doSomething(value)
}
```

Payable functions may have an arbitrary amount of parameters, but exactly one needs to be implicit and of a currency type. There may only be one function marked `@payable` in a contract.

Fallback

(Contract-specific.)

Fallback functions are another special kind of function, with a slightly modified declaration syntax:

```
public fallback() {
  // statements
}
```

Fallback functions should only contain 'simple' statements, just like initialisers. They are called whenever an attempt has been made to call a non-existent function of the containing contract. This may happen e.g. if the caller used an incorrect signature for the call. Oftentimes the Gas allocation for fallback execution is very low (2300), which only allows an event to be logged.

B.2.5 Structs

Structs in Flint are general-purpose constructs that group state and methods that can be used as self-contained blocks. They use the same syntax as defining constants and variables for properties. Structure methods are not protected as they can only be called by contract functions, and are required to be annotated `mutating` if they mutate the struct's state.

Declaration

The syntax of a struct declaration is:

```
struct <name> {
  // variables, constants, methods
}
```

Example:

```
struct Rectangle {
  var width: Int = 0
  var height: Int = 0

  func area() -> Int {
    return width * height
  }
}
```

Instances

The declaration of a struct only describes what types of variables it contains, what their initial values are, and what methods may be used to modify or access the struct data. To create concrete instances, each with individual data values, an instance has to be created, by calling the initialiser of a struct.

```
<struct-name>(<initialiser-parameter-values>)
```

> **Bug**
>
> Unlike for function calls, it is not required to write the labels for struct initialiser parameters.

Example:

```
let someRectangle: Rectangle = Rectangle()
```

When an instance is created, it is initialised with its initial values – in this case a width and height of `0`. This process can also be done manually using an initialiser. Defining initialisers is also required when default values are not specified for all properties of a struct. You can access the properties of the current struct with the special keyword `self`.

Example:

```
struct Rectangle {
  // Same definition as above with:
  public init(width: Int, height: Int) {
    self.width = width
    self.height = height
  }
}

let bigRectangle = Rectangle(width: 400, height: 10000)
```

Accessing properties/methods

Properties/methods of a struct instance can be accessed by writing the property/method name immediately after the instance identifier, separated by a period `.`:

```
<struct-instance>.<variable-name>
<struct-instance>.<constant-name>
<struct-instance>.<method-name>(<function-parameter-values>)
```

Examples:

```
bigRectangle.width // 400
bigRectangle.area() // evaluates to 4000000 by calling the `area` function
```

> **Planned feature**
>
> In the future a `static` keyword will be added to indicate struct functions which are callable without creating a specific instance. See #419.

Structs as function arguments

Structs can be passed by reference using the `inout` type modifier. The struct is then treated as an implicit reference to the value in the caller. Any modifications made to the struct will still be visible after the function is called.

When calling a function with an `inout` parameter, the given struct instance must be prefixed with `&` to indicate it is being passed by reference.

Example:

```
struct S {
  var x: Int

  init(x: Int) {
    self.x = x
  }
}

func foo() {
  let s: S = S(x: 8)

  byReference(s: &s)

  // Here s.x == 10

  // This is not supported:
  //byValue(s: s)

  // Here s.x == 10 would still be true.
}
```

```
func byReference(s: inout S) {
  s.x = 10
}

// This is currently not supported:
//func byValue(s: S) {
//  s.x = 12
//}
```

Planned feature

Passing structs by value (copying the struct into storage or memory) is a planned feature. See #133.

B.2.6 Contracts

Contracts lie at the heart of Flint. They are the core building blocks of a program's code. Constants and variables can be defined inside contracts to be stored in the Ethereum network.

Declaration

The declaration of a Flint contract consists of multiple parts. The properties are declared in a single block using the keyword `contract` followed by the contract name that will be used as the identifier.

```
contract <name> {
  // constant and variable declarations, event declarations
}
```

Example:

```
contract Bank {
  var owner: Address
  let name: String = "Bank"
  event Shutdown(reason: String)
}
```

Type states

Flint introduces the concept of type states. Insufficient and incorrect state management in Solidity code have led to security vulnerabilities and unexpected behaviour in widely deployed smart contracts. Avoiding these vulnerabilities by the design of the language is a strong advantage.

Type states of a contract represent the possible states it can be in. At any point of time, the contract on the network can only exist in a single state. Special `become`

statements can be used within functions to move the contract to a different type state.

A contract declaration may optionally include a list of its type states:

```
contract <name> (<type-state-1>, <type-state-2>, ...) {
  // constant and variable declarations, event declarations
}
```

Type states should be valid identifiers, starting with a capital letter.

Example:

```
contract Auction (Preparing, InProgress, Terminated) {}
```

Using type state protection, it is possible to specify that only certain functions will be callable when the contract is in a given type state.

Protection blocks

The remaining parts of a contract are its protection blocks. While traditional computer programs have an entry point (the `main` function), smart contracts do not. After a contract is deployed on the blockchain, its code does not run until an Ethereum transaction is received. Smart contracts are in fact more akin to RESTful web services presenting API endpoints. It is important to prevent unauthorised parties from calling sensitive functions.

In Flint, functions of a contract are declared within protection blocks, which restrict when the enclosed functions are allowed to be called.

There are two elements to protection blocks, the caller group and the optional type state protection (see type states for more detail).

A minimal protection block of contract `<contract-name>` with the caller group `<caller-group>` is declared as:

```
<contract-name> :: (<caller-group>) {
  // functions
}
```

The caller can optionally be captured into a variable (see caller group variable):

```
<contract-name> :: <variable> <- (<caller-group>) {
  // functions
}
```

The protection block can optionally also check that the contract is in a given type state (see type state protection):

```
<contract-name> @(<type-state>) :: (<caller-group>) {
  // functions
}
```

Alternatively, protection blocks can be declared within the contract declaration part with the same syntax but using `self` instead of the contract name:

```
contract <contract-name> {
  // ...
  self :: (<caller-group>) {
    // ...
  }
}
```

Solidity uses function modifiers to insert dynamic checks in functions, which can for instance abort unauthorised calls. However, it is easy to forget to specify these checks, as the language does not require programmers to write them.

Having a language construct which protects functions from invalid calls could require programmers to systematically think about which parties should be able to call the functions they are about to define.

In Flint, functions of a contract are declared within protection blocks, which protect the functions from invalid access.

Caller group

Caller groups consist of a list of caller members enclosed in parentheses. These caller members may be identified using multiple mechanisms, as listed below. Functions inside protection blocks can only be called by an Ethereum address (the 'caller' address) that satisfies at least one of the caller members of that protection block.

Name	Flint type	Callable when
Predicate function	`Address -> Bool`	The function is called with the caller as input, must return `true`.
0-ary function	`() -> Address`	The returned address must match the caller address.
State property (single address)	`Address`	The address property must match the caller address.
State property (list of addresses)	`[Address]` or `Address[n]`	The caller address must be contained within the list of addresses.
State property (dictionary of addresses)	`[T: Address]`	The caller address must be contained with in the values of the dictionary.
Any	`any`	Always.

Examples:

```
contract Bank {
  let owner: Address
  var managers: [Address]
}
```

```
Bank :: (owner, managers) {
  // ...
}

contract Lottery {}

Lottery :: (lucky) {
  func lucky(address: Address) -> Bool {
    // return true or false
  }
}
```

The Ethereum address of the caller of a function is unforgeable. It is not possible to impersonate another user, as a consequence of Ethereum's mechanism which generates public addresses from private keys. Transactions are signed using a private key, and determine the public key of the caller. Stealing a caller capability would hence require stealing a private key. The recommended way for Ethereum users to transfer their ability to call functions is to either change the backing address of the caller capability they have (the smart contract must have a function which allows this), or to securely send their private key to the new owner, outside of the Ethereum platform.

Calls to Flint functions are validated both at compile-time and runtime, with runtime checks only being added where necessary.

Caller group variable

It is sometimes useful to know which address initiated the current transaction, in addition to verifying it with caller groups. This is possible with the optional caller group variable.

```
<contract-name> :: <variable> <- (<caller-group>) {
  // functions
}
```

Example:

```
contract AddressBook {
  var book: [Address: String] = [:]
}

AddressBook :: address <- (any) {
  public func remember(name: String) {
    book[address] = name
  }
}
```

Type state protection

A protection block may also be used to ensure that certain functions are only called when the contract is in a given type state.

```
<contract-name> @(<type-state>) :: (<caller-group>) {
  // functions
}
```

Example:

```
contract Poll(Open, CountingVotes, Result) {
  // ...
}

Poll @(Open) :: (any) {
  public func voteFor(option: String) {
    // ...
  }
}
```

In this example the `voteFor` function could only be called when the `Poll` was in the `Open` state.

Static checking

In a Flint function, if a function call to another Flint function is performed, the compiler checks that the caller meets the caller protection.

Consider the following example:

```
Bank :: (any) {
  func foo() {
    // Error: Protection "any" cannot be used to perform a call to a
    // function for "manager"
    bar()
  }
}

Bank :: (manager) {
  func bar() {}
}
```

Within the context of `foo`, the caller is regarded as `any`. It is not certain that the caller also satisfies the `manager` protection, so the compiler rejects the call.

Dynamic checking

In the above example, it is still possible for `foo` to satisfy the protections of the function `bar`. For such cases, two additional language constructs exist:

- `try? bar()`: The function `bar` is called if, at runtime, the protections are satisfied (i.e. the caller satisfies the caller protection and the state of the contract satisfies the type state protection). The expression `try? bar()` returns a boolean if successful.
- `try! bar()`: If at runtime `bar` protections are not satisfied an exception is thrown (reverting the transaction) and the function is not executed.

Multiple protections

A contract behaviour declaration can be restricted by multiple caller protections. Consider the following contract behavior declaration:

```
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {}
}
```

The function `forManagerOrCustomers` can be called either by the manager, or by any of the accounts registered in the bank.

Calls to functions of multiple protections are accepted if **each** of the protections of the enclosing function are compatible with **any** of the target function's protections.

Consider the following examples:

```
// Insufficient protections
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {
    // Error: "accounts" is not compatible with "manager"
    forManager()
  }
}

Bank :: (manager) {
  func forManager() {}
}

// Sufficient protections
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {
    // Valid: "manager" is compatible with "manager", and "accounts" is
    // compatible with "accounts"
    forManagerOrCustomers2()
  }
}

Bank :: (accounts, manager) {
  func forManagerOrCustomers2() {}
}

// "any" is compatible with any caller protection
Bank :: (manager, accounts) {
  func forManagerOrCustomers() {
    // Valid: "manager" is compatible with "manager"
    // (and "any", too), and "accounts" is compatible with "any"
    forManagerOrCustomers2()
  }
}
```

```
// The caller protection "manager" has no effect:
// "any" is compatible with any caller protection
Bank :: (manager, any) {
  func forManagerOrCustomers2() {}
}
```

Visibility modifiers

Variables declared in the contract can have modifiers in front of their declaration which control the automatic synthesis of variable accessors and mutators. By the nature of smart contracts all storage is visible already, but providing accessors makes that process easier.

- `public` access synthesises an accessor and a mutator so that the storage variable can be viewed and changed by anyone.
- `visible` access synthesises an accessor to the storage variable which allows it to be viewed by anyone.
- `private` access means that nothing is synthesised (but both accessors and mutators can still be manually specified).

An accessor, if synthesised for variable `<name>` or type `<type>`, has the signature `public func get<Name>() -> <type>`. A mutator, if synthesised for the same variable, has the signature `public mutatic func set<Name>(to: <type>)`.

Example:

```
public var value: Int
visible var name: String = "Bank"
```

The above declarations cause these functions to be synthesised:

```
public func getValue() -> Int
public func setValue(to: Int)
public func getName() -> String
```

Events

JavaScript applications can listen to events emitted by an Ethereum smart contract.

In Flint, events are declared in contract declarations. They use a similar syntax to functions, except using the keyword `event`.

```
event <event-name>(<event-parameter-1>, <event-parameter-2>, ...)
```

Like functions, some of the parameters can have default values, but these must be declared at the end of the parameter list.

Events can then be emitted using the keyword `emit` followed by an event call. An event call is similar to a function call (parameters must be provided in order, and they must have the correct label and type; if any optional parameters are omitted, their default value will be used automatically).

```
contract Bank {
  var balances: [Address: Int]
  event CompletedTransfer(origin: Address, destination: Address, amount: Int)
}

Bank :: caller <- (any) {
  mutating func transfer(to: Address, value: Int) {
    // Note the following 2 lines are unsafe!
    balances[caller] -= value
    balances[to] += value

    // A JavaScript client could listen for this event:
    emit CompletedTransfer(origin: caller, destination: to, amount: value)
  }
}
```

B.2.7 Traits

Flint has the concept of 'traits', based in part on traits in the Rust language. Traits describe the partial behaviour of the contracts or structs which conform to them. For contracts, traits constitute a collection of functions, function signatures in protection blocks, and events. For structs, traits only constitute a collection of functions and function signatures.

Contracts or structs can conform to multiple traits. The Flint compiler enforces the implementation of function signatures in the trait and allows usage of the functions declared in them. Traits allow a level of abstraction and code reuse for contracts and structs.

> **Planned feature**
>
> In the future, the Flint standard library will include traits providing common functionality to contracts (`Ownable`, `Burnable`, `MultiSig`, `Pausable`, `ERC20`, `ERC721`, etc.) and structs (`Transferable`, `RawValued`, `Describable` etc.). It will also form the basis for allowing end users to access compiler level guarantees and restrictions as in assets and Numerics.

Struct traits

Traits can be declared for structs using the syntax:

```
struct trait <trait-name> {
  // trait members
}
```

Structs can conform to struct traits using the syntax:

```
struct <struct-name>: <trait-1>, <trait-2>, ... {
  // ...
}
```

Struct traits can contain functions, function signatures, initialisers, and initialiser signatures. A function or initialiser signature simply declares the name (for a function) and parameter types, without providing the actual code implementation.

Example:

In this example we define an `Animal` struct trait. The `Person` struct then conforms to the `Animal` trait.

```
struct trait Animal {
  // Must have an empty and named initialiser.
  public init()
  public init(name: String)

  // These are signatures that conforming structures must implement
  // access properties of the structure.
  func isNamed() -> Bool
  public func name() -> String
  public func noise() -> String

  // This is a pre-implemented function using the functions already in the trait.
  public func speak() -> String {
    if isNamed() {
      return name()
    }
    else {
      return noise()
    }
  }
}

struct Person: Animal {
  let name: String

  public init() {
    self.name = "John Doe"
  }
  public init(name: String) {
```

```
    self.name = name
  }

  // People always have a name, it's just not always known.
  func isNamed() -> Bool {
    return true
  }

  // These access the properties of the struct.
  public func name() -> String {
    return self.name
  }

  public func noise() -> String {
    return "Huh?"
  }

  // Person can also have functions in addition to Animal.
  public func greet() -> String {
    return "Hi"
  }
}
```

Contract traits

Traits can be declared for contracts using the syntax:

```
contract trait <trait-name> {
  // trait members
}
```

Contracts can conform to contract traits using the following syntax for their declaration part:

```
contract <contract-name>: <trait-1>, <trait-2>, ... {
  // ...
}
```

Contract traits can contain anonymous contract behaviour declarations containing functions, function signatures, and events.

Example:

In this example, we define `Ownable`, which declares a contract as something that can be owned and transferred. The `Ownable` trait is then used by the `ToyWallet` contract allowing the use of methods in `Ownable`. This demonstrates how we can expose contract properties:

```
contract trait Ownable {
  event OwnershipRenounced(previousOwner: Address)
  event OwnershipTransfered(previousOwner: Address, newOwner: Address)

  self :: (any) {
    public func getOwner() -> Address
  }

  self :: (getOwner) {
    func setOwner(newOwner: Address)

    public func renounceOwnership() {
      emit OwnershipRenounced(getOwner())
      setOwner(0x000...)
    }

    public func transferOwnership(newOwner: Address) {
      assert(newOwner != 0x000...)
      emit OwnershipTransfered(getOwner(), newOwner)
      setOwner(newOwner)
    }
  }
}

contract ToyWallet: Ownable {
  visible var owner: Address // visible automatically creates getOwner
  // Skipping initialiser not relevant for this example
}

ToyWallet :: (getOwner) {
  func setOwner(newOwner: Address){
    self.owner = newOwner
  }
}
```

External traits

Traits can be declared for external contracts using the syntax:

```
contract trait <trait-name> {
  // trait members
}
```

See the specifying the interface of external calls section for more information.

Polymorphic self

`Self` (note the capital 'S') is a special type available only in struct and contract traits. It refers to the type that implements the current trait, but not any other type that conforms to that trait. This is particularly useful when providing default implementations for functions in a trait. See assets for an example in the standard library.

Example without `Self`:

```
struct trait Unit {
  func add(source: inout Unit)
}

struct Metre: Unit {
  var length: Int = 0
  func add(source: inout Unit) {
    length += source.length // compilation error here
  }
}
```

In the above example, we only want to be able to add metres to metres. Accessing `source.length` is invalid, because `length` is only declared in `Metre`. Instead, using `Self`:

```
struct trait Unit {
  mutating func add(source: inout Self)
}

struct Metre: Unit {
  var length: Int = 0
  mutating func add(source: inout Metre) {
    length += source.length
  }
}

struct Litre: Unit {
  var volume: Int = 0
  mutating func add(source: inout Litre) {
    volume += source.volume
  }
}
```

In this example, both `Metre` and `Litre` are valid. But a call like:
`aMetreInstance.add(source: &aLitreInstance)`
would cause a compile-time error.

B.2.8 Expressions

Expressions are at the core of any computation done in Flint code. Evaluating an expression results in a single value of a given type. Expressions can be nested to arbitrary layers.

The expressions available in Flint are:

- Literal (e.g. `1`, `"hello"`, `false`, etc.) – constant value; see literals.
- Range (e.g. `<expr-1>..<expr-2>`, `<expr-1>...<expr-2>`) – see ranges.
- Binary expression (e.g. `<expr-1> <op> <expr-2>`) – a binary operation `<op>` applied to the expressions `<expr-1>` and `<expr-2>`; see operators.
- Struct reference (e.g. `&<expr>`) – see structs as function arguments.
- Function call (e.g. `<function-name>(<param-1>: <expr-1>, <param-2>: <expr-2>, ...)`) – call to the function `<function-name>` with the results of the given expressions `<expr-1,2,...>` as parameters. See function calls.
- Dot access (e.g. `<expr-1>.<field>`) - access to the `<field>` field (variable, constant, function) or the result of `<expr-1>`.
- Index / key access (e.g. `<expr-1>[<expr-2>]`) – access to the given key of a list or dictionary.
- External call (e.g. `call <external-contract>.<function-name>(<param-1>: <expr-1>, ...)`) – call to the function of an external contract; see external calls.
- Type cast (e.g. `<expr> as! <type>`) – forced cast of the result of `<expr>` to `<type>`; see casting to and from Solidity types.
- Attempt (e.g. `try? <call>`, `try! <call>`) – attempt to call a function in a different protection block, see dynamic checking.

Function calls

Functions can then be called from within a contract protection block with the same identifier. The call arguments must be provided in the same order as the one they are declared in (in the function signature), and they must be labeled accordingly (the exception for this is struct initialisers). If any of the optional parameters are not provided, then their default values are going to be used automatically.

B.2.9 Literals

Literals represent fixed values in the source code that can be assigned to constants and variables.

Integer literals

Integer literals (Flint type `Int`) can be written as decimal numbers. The size of the `Int` type in Flint is 256 bits (32 bytes), so the highest allowed integer is quite large (`2^256 - 1`, more than 76 decimal degits). Underscores can be used to separate digits of integer literals.

Examples:

```
42
2019
1_22_333
1_000_000_000_000_000_000_000_000_000
```

Address literals

Address literals (Flint type `Address`) are written as 40 hexadecimal digits prefixed by a `0x`. Addresses are an important concept in Ethereum, referring to other contracts and accounts. Underscores can be used to separate digits of address literals.

Examples:

```
0x1234123412341234123412341234123412341234
0x0CB1DB10A4820BD10823AE0101F02198CAFEBABE
0xCAFEBABE_CAFEBABE_CAFEBABE_CAFEBABE_CAFEBABE
```

Boolean literals

Boolean literals (Flint type `Bool`) are simply `true` and `false`.

String literals

String literals (Flint type `String`) are sets of characters enclosed in double quotes `"..."`.

Examples:

```
""
"hello"
"This is a sentence."
```

> **Bug**
>
> Due to the fact that `Strings` are currently stored in a single EVM memory slot, they cannot be longer than 32 bytes. See #133.

List literals

List literals (Flint type `[T]` or `T[n]` for some Flint type `T`) currently only include the empty list `[]`.

> **Planned feature**
>
> In the future, Flint will have non-empty list literals written as `[x, y, z, ...]` where `x`, `y`, `z`, etc. are literals of type `T`. See #420.

Dictionary literals

Dictionary literals (Flint type `[T: U]` for some Flint types `T` and `U`) currently only include the empty dictionary `[:]`.

> **Planned feature**
>
> In the future, Flint will have non-empty dictionary literals written as `[x: a, y: b, z: c, ...]` where `x`, `y`, `z`, etc. are literals of type `T` and `a`, `b`, `c`, etc. are literals of type `U`. See #420.

Self

The special keyword `self` refers to the current struct instance or contract containing the current function.

B.2.10 Operators

An operator is a special symbol used to check, change, or combine values. Flint supports common Swift operators and attempts to eliminate common coding errors.

Arithmetic operators

Flint supports the following arithmetic operators for `Int` expressions:

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication

- `/` - Division
- `**` - Exponentiation

Examples:

```
1 + 2 // equals 3
5 - 3 // equals 2
2 * 3 // equals 6
10 / 2 // equals 5
2 ** 3 // equals 8
```

Flint has unique safe arithmetic. The `+`, `-`, `*` and `**` operators throw an exception and abort execution of the smart contract when an overflow occurs. The `/` operator implements integer division. No underflows can occur as floating-point numbers are not supported yet. The performance overhead of the safe operators is low.

In rare cases, allowing overflows is the intended behaviour. Flint also supports overflowing operators, which will not crash on overflow:

- `&+` - Unsafe addition
- `&-` - Unsafe subtraction
- `&*` - Unsafe multiplication

Boolean operators

These operators all result in `Bool`:

- `==` - Equal to
- `!=` - Not equal to
- `||` - Logical or
- `&&` - Logical and
- `<` - Less than
- `<=` - Less than or equal to
- `>` - Greater than
- `>=` - Greater than or equal to

Examples:

```
1 == 1 // true because 1 is equal to 1
2 != 1 // true because 2 is not equal to 1
2 > 1 // true because 2 is greater than 1
1 < 2 // true because 1 is less than 2
1 >= 1 // true because 1 is greater than or equal to 1
2 <= 1 // false because 2 is not less than or equal to 1
true || false // true because one of true and false is true
true && false // false because one of true and false is false
```

B.2.11 Statements

Statements control the execution of code in a function, enable looping, conditional behaviour, and more.

Variable/constant declaration and assignment

Declaration of variables and constants is a statement (see variables and constants). Syntax:

```
let <name>: <type> = <expression>
let <name>: <type>
var <name>: <type> = <expression>
var <name>: <type>
```

Compound assignment

Flint also provides compound assignment statements that combine assignment (=) with another operator. Namely:

- `+=` Compound addition
- `-=` Compound subtraction
- `*=` Compound times
- `/=` Compound division

Example:

```
x += 5
// is equivalent to:
x = x + 5
```

Loops

`for-in` loops can be used to iterate over sequence. Currently this supports lists, dictionary values and ranges. Syntax:

```
for let <variable-name>: <type> in <sequence> {
  // ...
}
```

Alternatively, the iteration value can be a variable, so it can be modified, though modifications are reset on each loop:

```
for var <variable-name>: <type> in <sequence> {
  // ...
}
```

Example:

Assuming a variable-length list `names` (of type `[String]`), it can be iterated over, binding the current iteration value to the constant `name` of type `String`, using:

```
for let name: String in names {
  // do something with `name`
}
```

Conditionals

The `if` statement allows executing different code based on the result of a condition (of Flint type `Bool`). Syntax:

```
if <condition> {
  // ...
}
```

Example:

```
if x == 2 {
  // ...
}
```

Else clauses

The `if` statement can also provide an alternative set of statements known as an `else` clause which gets executed when the condition evaluates to `false`. Syntax:

```
if <condition> {
  // ...
} else {
  // ...
}
```

Example:

```
if x == 2 {
  // ...
} else {
  // ,,,
}
```

Become statements

(Contract-specific.)

The `become` statement can be used to change the type state (see type states) of the current contract. The execution of code is terminated after a `become` statement is executed, and the contract will then transition to the specified type state. Syntax:

```
become <type-state>
```

Example:

```
contract Semaphore(Red, Green) {}

Semaphore @(Red) :: (any) {
  public func wait() {
    become Green
  }
}

Semaphore @(Green) :: (any) {
  public func wait() {
    become Red
  }
}
```

Return statements

A `return` statement can be used to provide the output value of a function with a declared return type (see functions). Syntax:

```
return <expression>
```

Example:

```
Semaphore @(Red) :: (any) {
  public func countWaitingCars() -> Int {
    return 200
  }
}
```

Do-catch blocks

`do-catch` blocks can be used to handle errors in execution in a controlled manner. Currently, the only supported error is an external call error (see external calls). Syntax:

```
do {
  // ...
} catch is ExternalCallError {
  // ...
}
```

B.2.12 External calls

External calls refer to a Flint contract calling the functions of other contracts deployed on the Ethereum network. They also allow money to be transferred from Flint contracts to other accounts and contracts, enabling full participation in the Ethereum network.

However, external contracts include their own set of possible risks and security considerations. When writing code that interacts with external contracts, it is important to keep in mind that:

1. External contracts may execute arbitrary code when called – although the called contract does not have access to the memory or state storage of the calling (Flint) contract, it may still cause problems. In particular, care should be taken when handling the output returned from an external contract. Additionally, the external contract may call arbitrary function of the calling (Flint) contract, potentially resulting in a re-entrancy attack.

2. Interfaces of external contracts may be incorrectly specified – since the EVM does not retain any type information, it is up to the programmer to correctly specify the functions available on an external contract. If the interface is specified incorrectly, this may lead to the wrong function being called and money being lost.

   Planned feature

   In the future, external calls will include automatic re-entrancy attack protection, where no function of a Flint contract will be callable during the execution of an external call. See #74.

Specifying the interface

The interface of an external contract is specified using a special `external` trait. Syntax:

```
external trait <trait-name> {
  // functions
}
```

The functions declared inside an external trait may not include any modifiers, and their parameters and return types (if used) must be specified using Solidity types.

Currently, deploying contracts from within Flint code is not supported, so neither initialisers nor fallbacks can be provided in external traits.

Example:

```
external trait ExternalBank {
  @payable
  func pay() -> int256
  func withdraw(amount: int256) -> int256
}
```

Creating an instance

To work with an external contract in a type-safe manner, every external trait automatically creates an implicit constructor, which takes a single `address` parameter.

```
<external-trait-name>(address: <address>)
```

Example:

```
external trait Ext {}
contract X {}

X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
  }
}
```

Calling functions

Functions of an external contract instance may be called using the keyword `call`. Flint provides two modes of operation for external calls, and they are semantically similar to `try` in Swift.

```
call <contract>.<function-name>(<parameters>)
call! <contract>.<function-name>(<parameters>)
```

The forced mode is invoked with the syntax `call!` (note the exclamation mark). If the external call fails for any reason (e.g. the external contract runs out of gas), the entire transaction will revert.

```
X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
    call! extInstance.someFunction()
  }
}
```

The default (safe) mode is invoked with the syntax `call` (without the exclamation mark). Any default call must be inside a `do-catch` block, and a failure in the external call will cause the code in the `catch` block to be executed.

```
X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
    do {
      call extInstance.someFunction()
    } catch is ExternalCallError {
```

```
      // handle the error here
    }
  }
}
```

Planned feature

A third mode will be available in the future, `call?`. It will return an `Optional` type, like in Swift, intended to be used with the (also planned) `if let ...` construct. See #140.

Specifying hyper-parameters

In addition to function parameters, there are two more 'hyper-parameters' that need to be set when performing an external call.

The `gas` hyper-parameter (defaults to `2300`) with type `Int` specifies how much Gas is allocated for the external call. Executing any code in EVM costs Gas and so the more Gas is provided, the more work can be done in a contract. The default amount, `2300`, is enough to emit a single event (at the time of writing).

The `value` hyper-parameter (defaults to `0`) with type `Wei` specifies how much, if any, Wei is attached to the external call. Providing a non-zero amount causes money to be transferred from the calling (Flint) contract to the external contract. `value` must be specified if and only if calling a function marked as `@payable`.

To specify `gas` and/or `value` for an external call, the syntax is:

```
call(<hyper-parameters>) <contract>.<function-name>(<parameters>)
call(<hyper-parameters>)! <contract>.<function-name>(<parameters>)
```

Example:

```
X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
    call(gas: 10000)! extInstance.someLongFunction()
    call(value: Wei(unsafeRawValue: 100))! extInstance.someExpensiveFunction()
    call(
        gas: 10000, value: Wei(unsafeRawValue: 100)
      )! extInstance.someLongExpensiveFunction()
  }
}
```

Casting to and from Solidity types

Since the types of external contract function parameters and return values are specified using Solidity types, values must be converted before they are used for an external call. This is facilitated using the type casting expression.

Example:

```
X :: (any) {
  public func callback(externalAddress: Address) {
    let extInstance = Ext(address: externalAddress)
    var flintInt: Int = 1
    call! extInstance.someFunctionTakingAnInt(someParameter: flintInt as! int256)

    flintInt = (call! extInstance.someReturningFunction()) as! Int
  }
}
```

The forced cast (`as!`) expression converts Flint types to Solidity types and vice versa, after performing some basic runtime checks to make sure that the original value fits into the target value, since Solidity supports integer types of smaller sizes than the Flint default of 256 bits. An error results in the transaction being reverted.

> **Planned feature**
>
> In the future, casting failures will be possible to handle using `do-catch` blocks or an optional cast mode `as?`.

B.2.13 Enumerations

An enumeration defines a common group of values with the same type and enables working with those values in a type-safe way within Flint code. The syntax is:

```
enum <name>: <associated-type> {
  case <case-name>
  // additional cases...
}
```

Example:

```
enum CompassPoint: Int {
  case north
  case south
  case east
  case west
}
```

The values defined in an enumeration (such as `north`, `south`, `east` and `west`) are its enumeration cases. Each enumeration defines a new user-defined type. To access a given case, dot syntax is used:

```
<enum-name>.<case-name>
```

Example:

```
var direction: CompassPoint
direction = CompassPoint.north
```

Associated values

You can assign raw values to enumeration cases. The values need to match the type associated with the enumeration. Flint will also try to infer the raw value of cases by default based on the raw value of the last declared enumeration case.

Example:

```
enum Numbers: Int {
  case one = 1
  case two = 2
  case three // Numbers.three == 3
  case four // Numbers.four == 4
}
```

B.3 Standard library

B.3.1 Assets

Numerous attacks targeting smart contracts, such as ones relating to re-entrancy calls (see TheDAO), allowed hackers to steal a contract's Ether. Some of these happened because smart contracts encoded Ether values as integers, making it easy to make mistakes when performing Ether transfers between variables, or to forget to record Ether arriving or leaving the smart contract.

Flint supports special safe operations when handling assets, such as Wei (the smallest unit of Ether). They help ensure the contract's state consistently represents its Wei value, preventing attacks such as TheDAO.

A simple use of Wei:

```
contract Wallet {
  var owner: Address
  var contents: Wei = Wei(unsafeRawValue: 0)
}

Wallet :: caller <- (any) {
  public init() {
    owner = caller
  }

  @payable
  public mutating func deposit(implicit value: Wei) {
    // Record the Wei received into the contents state property.
    // Value is passed by reference.
    contents.transfer(source: &value)
  }
}
```

```
Wallet :: (owner) {
  public mutating func withdraw(value: Int) {
    // Transfer an amount of Wei into a local variable. This
    // removes Wei from the contents state property.
    var w: Wei = Wei(source: &contents, amount: value)

    // Send Wei to the owner's Ethereum address.
    send(address: owner, value: &w)
  }

  public func getContents() -> Int {
    return contents.getRawValue()
  }
}
```

Another example which uses Wei is the Bank example.

```
contract Wallet {
  var beneficiaries: [Address: Wei]
  var weights: [Address: Int]
  var bonus: Wei
  var owner: Address
}

Wallet :: (any) {
  @payable
  mutating func receiveBonus(implicit newBonus: inout Wei) {
    bonus.transfer(source: &newBonus)
  }
}

Wallet :: (owner) {
  mutating func distribute(amount: Int) {
    let beneficiaryBonus = bonus.getRawValue() / beneficiaries.count
    for let person: Address in beneficiaries {
      var allocation = Wei(source: &balance, amount: amount * weights[person])
      allocation.transfer(source: &bonus, amount: beneficiaryBonus)
      send(address: beneficiaries[i], value: &allocation)
    }
  }
}
```

Wei is an example of an asset, and it is a struct conforming to the `struct trait Asset`, available in the standard library. It is possible to declare custom structs which will behave like assets:

```
struct MyWei : Asset {
  var rawValue: Int = 0

  init(unsafeRawValue: Int) {
    self.rawValue = unsafeRawValue
  }

  init(source: inout MyWei, amount: Int) {
    transfer(source: &source, amount: amount)
  }

  init(source: inout MyWei) {
    let amount: Int = source.getRawValue()
    transfer(source: &source, amount: amount)
  }

  mutating func setRawValue(value: Int) -> Int {
    rawValue = value
    return rawValue
  }

  func getRawValue() -> Int {
    return rawValue
  }
}
```

The `transfer` functions are declared in the `Asset` trait and are inherited automatically. For the time being, traits do not support default implementations for initialisers or variables, so custom assets have to include the code above. Struct traits also do not support variables as part of the conformance, which is why `setRawValue` and `getRawValue` are required.

B.3.2 Global functions

Global functions in the standard library are special function which can be called from any contract, struct, or contract group.

Assertions

Assertions are checks that happen at runtime. They are used to ensure an essential condition is satisfied before executing any further code. If the boolean condition evaluates to `true` then the execution continues as usual. Otherwise the transaction is reverted.

```
assert(<expr>)
```

Example:

```
assert(x == 2)
```

In essence an assertion is a shorthand for the longer:

```
if x == 2 {
  fatalError()
}
```

Fatal error

`fatalError()` is a function exposed that reverts a transaction when called. This means that any contract storage changes are rolledback and no values are returned.

Send

`send(address: Address, value: inout Wei)` sends the `value` Wei to the Ethereum address `address`, and clears the contents of `value`. It is a simpler way to perform a money transfer compared to external calls, but does not allow hyper-parameters to be specified.

Annex C

Example Real-World Contract

In this appendix, we present a somewhat realistic distributed application using the new external calls and error handling features introduced in Flint.

The application consists of two contracts: a travel agency and a number of weather stations.

C.1 Weather Station

Each weather station has a name and its owner can record readings of weather (sunny, rain, snow), which are saved in the station contract state storage. The latest weather reading can be read by users and other contracts by calling the function `getLatestReading`.

```
enum WeatherReading: Int {
  case none
  case sunny
  case rain
  case snow
}

contract WeatherStation {
  var owner: Address
  visible var name: String
  var latestReading: WeatherReading
}

WeatherStation :: caller <- (any) {
  public init(newName: String) {
    self.owner = caller
    self.name = newName
    latestReading = WeatherReading.none
  }
```

```
  // Note that this function returns String,
  // not WeatherReading, because Solidity ABI does not
  // support custom enumeration types, and there are
  // currently no functions to convert to and from
  // raw values of enums in Flint.
  public func getLatestReading() -> String {
    var reading: String = "none"
    if latestReading == WeatherReading.sunny {
      reading = "sunny"
    }
    if latestReading == WeatherReading.rain {
      reading = "rain"
    }
    if latestReading == WeatherReading.snow {
      reading = "snow"
    }
    return reading
  }
}

WeatherStation :: (owner) {
  public mutating func recordSunny() {
    self.latestReading = WeatherReading.sunny
  }

  public mutating func recordRain() {
    self.latestReading = WeatherReading.rain
  }

  public mutating func recordSnow() {
    self.latestReading = WeatherReading.snow
  }

  public mutating func transferOwneship(newOwner: Address) {
    owner = newOwner
  }

  public mutating func changeName(newName: String) {
    name = newName
  }
}
```

C.2 Travel Agency

The weather stations are then used by a 'travel agency'. The owner of the agency can add multiple weather stations to the agency's `locations`. Anyone on the network can then call `recommendLocation`, which will cause the agency contract to query each weather station in turn for its weather reading using an external call, then return a location that has sunny weather and fewest clients.

```
external trait WeatherStation {
  func getName() -> string
  func getLatestReading() -> string
}

contract TravelAgency {
  var locations: [Address] = []
  var locationCount: Int = 0
  var clients: [Address: Int] = [:]
  visible var owner: Address
  event Adding(location: Address)
}

TravelAgency :: caller <- (any) {
  public init() {
    self.owner = caller
  }

  public func getLocationCount() -> Int {
    return locations.size
  }

  func checkLocation(location: Address, recommended: Address) -> Bool {
    let station: WeatherStation = WeatherStation(address: location)
    var goodLocation: Bool = false
    do {
      let reading: String = (call station.getLatestReading()) as! String
      if reading == "sunny" {
        if (recommended == 0x0000000000000000000000000000000000000000)
            || (clients[location] < clients[recommended]) {
          goodLocation = true
        }
      }
    } catch is ExternalCallError {}
    return goodLocation
  }

  public func getClientsAt(location: Address) -> Int {
```

```
    return clients[location]
  }

  mutating func addClientAt(location: Address) {
    clients[location] += 1
  }

  public mutating func recommendLocation() -> Address {
    var recommendedName: String = "no good locations today!"
    var recommendedLocation: Address = 0x0000000000000000000000000000000000000000
    for let location: Address in locations {
      do {
        let station: WeatherStation = WeatherStation(address: location)
        if checkLocation(location: location, recommended: recommendedLocation) {
          recommendedName = (call station.getName()) as! String
          // only set if the call didn't throw
          recommendedLocation = location
        }
      } catch is ExternalCallError {
        // station not working properly, ignore
      }
    }
    emit Adding(location: recommendedLocation)
    if recommendedLocation != 0x0000000000000000000000000000000000000000 {
      addClientAt(location: recommendedLocation)
    }
    return recommendedLocation
  }
}

TravelAgency :: (owner) {
  public mutating func addLocation(location: Address) {
    clients[location] = 0
    locations[locationCount] = location
    locationCount += 1
  }

  public mutating func transferOwneship(newOwner: Address) {
    owner = newOwner
  }

  public mutating func startNextDay() {
    for let location: Address in locations {
      clients[location] = 0
    }
```

```
  }
}
```

In this application, external calls are used for passing data from one contract to another. This is meant to resemble a service-oriented architecture found in real-world distributed applications.