Solang Solidity Compiler

Release 0.1.6

Jan 25, 2021

Contents:

Welcome to the Solang Solidity Compiler, the portable Solidity compiler. Using Solang, you can compile smart contracts written in Solidity for Parity Substrate, Solana, Sawtooth Sabre, and Ethereum ewasm. It uses the llvm compiler framework to produce WebAssembly (wasm) or BPF contract code. As result, the output is highly optimized, which saves you in gas costs.

Solang aims for source file compatibility with the Ethereum EVM Solidity compiler, version 0.7. Where differences exists, this is noted in the *language documentation*. Also, check our *Status* page. The repository can be found on github and we have a channel on chat.hyperledger.org.

CHAPTER 1

Installing Solang

The Solang compiler is a single binary. It can be installed in different ways.

1.1 Download release binaries

For Linux x86-64, there is a binary available in the github releases:

https://github.com/hyperledger-labs/solang/releases/download/v0.1.6/solang_linux

For Windows x64, there is a binary available:

https://github.com/hyperledger-labs/solang/releases/download/v0.1.6/solang.exe

For MacOS, there is a binary available:

https://github.com/hyperledger-labs/solang/releases/download/v0.1.6/solang_mac

1.2 Using hyperledgerlabs/solang docker hub images

New images are automatically made available on docker hub. There is a release *v0.1.6* tag and a *latest* tag:

```
docker pull hyperledgerlabs/solang
```

The Solang binary is stored at `/usr/bin/solang` in this image. The *latest* tag gets updated each time there is a commit to the main branch of the Solang git repository.

1.3 Build Solang using Dockerfile

First clone the git repo using:

```
git clone https://github.com/hyperledger-labs/solang
```

Then you can build the image using:

```
docker image build .
```

1.4 Building Solang from source

In order to build Solang from source, you will need rust 1.43.0 or higher, and a build of llvm based on our tree. There are a few patches which are not upstream yet First, follow the steps below for installing llvm and then proceed from there.

If you do not have the correct version of rust installed, go to rustup.

1.5 Installing the LLVM Libraries

Solang needs a build of llvm with some extra patches. You can either download the pre-built binaries or build your own from source. After that, You need to add the *bin* directory to your path, so that the build system of Solang can find the correct version of llvm to use.

1.5.1 Installing LLVM on Linux

A pre-built version of llvm, specifically configured for Solang, is available at https://solang.io/download/llvm10.0-linux.tar.gz. This version is built using the dockerfile for building llvm on linux. After downloading, untar the file in a terminal and add it to your path.

```
tar zxf llvm10.0-linux.tar.gz
export PATH=$(pwd)/llvm10.0/bin:$PATH
```

1.5.2 Installing LLVM on Windows

A pre-built version of llvm, specifically configured for Solang, is available at https://solang.io/download/llvm10.0-win.zip. This version is built using the dockerfile for building llvm on Windows.

If you want to use the dockerfile yourself rather than download the binaries above, then this requires Docker Desktop installed, and then switched to windows containers. The result will be an image with llvm compressed in the file `c:\llvm10.0-win.zip`. Docker on Windows needs Hyper-V enabled. If you are running Windows 10 in a virtual machine, be sure to check this blog post.

After unzipping the file, add the bin directory to your path.

```
set PATH=%PATH%;C:\llvm10.0\bin
```

1.5.3 Installing LLVM on Mac

A pre-built version of llvm, specifically configured for Solang, is available on https://solang.io/download/llvm10.0-mac.tar.gz. This version is built with the instructions below. After downloading, untar the file in a terminal and add it to your path.

```
tar zxf llvm10.0-mac.tar.gz
xattr -rd com.apple.quarantine llvm10.0
export PATH=$(pwd)/llvm10.0/bin:$PATH
```

1.5.4 Building LLVM from source

The llvm project itself has a guide to installing from source which you may need to consult. First if all clone our llvm repository:

```
git clone git://github.com/seanyoung/llvm-project
cd llvm-project
```

Now switch to the bpf branch:

```
git checkout bpf
```

Now run cmake to create the makefiles. Replace the *installdir* argument to `CMAKE_INSTALL_PREFIX` with with a directory where you would like to have llvm installed, and then run the build:

```
cmake -G Ninja -DLLVM_ENABLE_ASSERTIONS=On '-DLLVM_ENABLE_PROJECTS=clang;lld'  \
        -DLLVM_ENABLE_TERMINFO=Off -DCMAKE_BUILD_TYPE=Release \
        -DCMAKE_INSTALL_PREFIX=installdir -B build llvm
cmake --build build --target install
```

Once the build has succeeded, the *installdir*/bin has to be added to your path so the Solang build can find the `llvm-config` from this build:

```
export PATH=installdir/bin:$PATH
```

And on Windows, assuming *installdir* was `C:\Users\User\solang-llvm`:

```
set PATH=%PATH%;C:\Users\User\solang-llvm\bin
```

1.6 Building Solang from crates.io

The latest Solang release is on crates.io. Once you have the correct llvm version in your path, simply run:

```
cargo install solang
```

1.7 Building Solang from git

Once you have the correct llvm version in your path, simply run:

```
git clone https://github.com/hyperledger-labs/solang/
cd solang
cargo build --release
```

The executable will be in `target/release/solang`.

CHAPTER 2

Status

2.1 Solidity Language Status

Solang is source compatible with the Ethereum Foundation Solidity Compiler version 0.7, with some caveats. Many language features have only recently been implemented, and have many unit tests. As with any new project, bugs are possible. Please report any issues you may find to github.

Differences:

- `immutable` is not supported. Note this is impossible to implement on any than chain other than Ethereum; this is purely an ethereum feature
- libraries are always statically linked into the contract code
- Solang generates WebAssembly or BPF rather than EVM. This means that the `assembly {}` statement using EVM instructions is not supported

Unique features to Solang:

- Solang can target different blockchains and some features depending on the target. For example, Parity Substrate uses a different ABI encoding and allows constructors to be overloaded.
- Events can be declared outside of contracts
- Base contracts can be declared in any order
- There is a `print()` function for debugging
- Strings can be formatted with python style format string, which is useful for debugging: `print("x = {}".format(x));`

2.2 Target Status

2.2.1 Parity Substrate

Solang works with Parity Substrate 2.0. This target is the most mature and has received the most testing so far.

2.2.2 Solana

Solang has a new target for Solana. This is in early stages right now, however it is under active development.

2.2.3 ewasm

ewasm has been tested with Hyperledger Burrow. Please use the latest master version of burrow, as ewasm support is still maturing in Burrow.

Some language features have not been fully implemnented yet on ewasm:

- The built in function `abi.encode()`, `abi.encodeWithSelector()`, `abi.encodeWithSignature()`, and `abi.encodePacked()`
- Contract storage variables types `string` and `bytes` are not implemented

2.2.4 Hyperledger Sawtooth

This is merely a proof-of-concept target, and has seen very little testing. On sawtooth, many Solidity concepts are impossible to implement:

- Return values from contract calls
- Calling other contracts
- Value transfers
- Instantiating contracts

CHAPTER 3

Running Solang

The Solang compiler is run on the command line. The solidity source file names are provided as command line arguments; the output is an optimized wasm or bpf file which is ready for deployment on a chain, and an abi file.

The following targets are supported right now: Parity Substrate, Solana Ethereum ewasm, and Sawtooth Sabre.

3.1 Using Solang on the command line

Usage:

> solang [OPTIONS]... [SOLIDITY SOURCE FILE]...

This means that the command line is `solang` followed by any options described below, followed by one or more solidity source filenames.

Options:

-v, –verbose Make the output more verbose. The compiler tell you what contracts have been found in the source, and what files are generated. Without this option Solang will be silent if there are no errors or warnings.

–target *target* This takes one argument, which can either be `ewasm`, `sabre`, `solana`, or `substrate`. The default is substrate.

–doc Generate documentation for the given Solidity files as a single html page. This uses the doccomment tags. The result is saved in `soldoc.html`. See *Tags* for further information.

-o, –output *directory* This option takes one argument, which is the directory where output should be saved. The default is the current directory.

-O *optimization level* This takes one argument, which can either be `none`, `less`, `default`, or `aggressive`. These correspond to llvm optimization levels.

–importpath *directory* When resolving `import` directives, search this directory. By default `import` will only search the current directory. This option can be specified multiple times and the directories will be searched in the order specified.

–help, -h This displays a short description of all the options

–standard-json This option causes Solang to emulate the behaviour of Solidity standard json output. No output files are written, all the output will be in json on stdout.

This feature is used by Hyperledger Burrow's deploy tool.

–emit *phase* This option is can be used for debugging Solang itself. This is used to output early phases of compilation.

Phase:

ast Output Abstract Syntax Tree, the parsed and resolved input

cfg Output control flow graph.

llvm-ir Output llvm IR as text.

llvm-bc Output llvm bitcode as binary file.

object Output wasm object file; this is the contract before final linking.

3.1.1 Running Solang from docker image

First pull the last Solang image from docker hub:

```
docker pull hyperledgerlabs/solang
```

And if you are using podman:

```
podman image pull hyperlederlabs/solang
```

Now you can run Solang like so:

```
docker run --rm -it hyperledgerlabs/solang --version
```

Or podman:

```
podman container run --rm -it hyperledgerlabs/solang --version
```

If you want to compile some solidity files, the source file needs to be available inside the container. You can do this via the -v command line. In this example `/local/path` should be replaced with the absolute path to your solidity files:

```
docker run --rm -it -v /local/path:/sources hyperledgerlabs/solang -o /sources /
↪sources/flipper.sol
```

On podman you might need to add `:Z` to your volume argument if SELinux is used, like on Fedora. Also, podman allows relative paths:

```
podman container run --rm -it -v .:/sources:Z hyperledgerlabs/solang -o /sources /
↪sources/flipper.sol
```

On Windows, you need to specify absolute paths:

```
docker run --rm -it -v C:\Users\User:/sources hyperledgerlabs/solang -o /sources /
↪sources/flipper.sol
```

3.2 Using Solang with Substrate

Solang builds contracts for Substrate by default. There is an solidity example which can be found in the examples directory. Write this to flipper.sol and run:

```
solang --target substrate flipper.sol
```

Now you should have a file called `flipper.contract`. The file contains both the ABI and contract wasm. It can be used directly in the Polkadot UI, as if the contract was written in ink!.

3.3 Using Solang with Solana

The Solana target is new and is limited right now, not all types are implemented and other functionality is incomplete. However, the flipper example can be used.

```
solang --target solana flipper.sol -v
```

This will produce two files called *flipper.abi* and *flipper.so*. The first is an ethereum style abi file and the latter being the ELF BPF shared object which can be deployed on Solana.

Solana has execution model which allows one program to interact with multiple accounts. Those accounts can be used for different purposes. In Solang's case, each time the contract is executed, it needs two accounts. The first account is for the *return data*, i.e. either the ABI encoded return values or the revert buffer. The second account is to hold the contract storage variables.

The output of the compiler will tell you how large the second account needs to be. For the *flipper.sol* example, the output contains *"info: contract flipper uses exactly 9 bytes account data"*. This means the second account should be exactly 9 bytes; anything larger is wasted. If the output is *"info: contract store uses at least 168 bytes account data"* then some storage elements are dynamic, so the size depends on the data stored. For example there could be a `string` type, and storage depends on the length of the string. The minimum is 168 bytes, but storing any non-zero-length dynamic types will fail.

If either account is too small, the transaction will fail with the error *account data too small for instruction*.

Before any function on a smart contract can be used, the constructor must be first be called. This ensures that the constructor as declared in the solidity code is executed, and that the contract storage account is correctly initialized. To call the constructor, abi encode (using ethereum abi encoding) the constructor arguments, and pass in two accounts to the call, the 2nd being the contract storage account.

Once that is done, any function on the contract can be called. To do that, abi encode the function call, pass this as input, and provide two accounts on the call. The second account must be the same contract storage account as used in the constructor. If there are any return values for the function, they are stored in the first return data account. The first 8 bytes is a 64 bits length, followed by the data itself. You can pass this into an ethereum abi decoder to get the expected return values.

There is an example of this written in node.

3.4 Using Solang with Sawtooth Sabre

When using Solang on Sawtooth Sabre, the constructor and function calls must be encoded with Ethereum ABI encoding. This can be done in different ways. In this guide we use ethabi. This can be installed using cargo:

```
cargo install ethabi-cli
```

In order to abi encode the calls, we need the abi for the contract. Let's compile flipper.sol for Sabre:

```
solang --target sabre --verbose flipper.sol
```

We now have a file `flipper.wasm` and `flipper.abi`. To deploy this, we need to create the constructor ABI encoding. Unfortunately ethabi already falls short here; we cannot encode constructor calls using the cli tools. However we can work round this by specify the constructor arguments explicitly. Note that if the constructor does not take any arguments, then the constructor data should be empty (0 bytes). So, since the constructor in flipper.sol takes a single bool, create it like so:

```
ethabi encode params -v bool true | xxd -r -p > constructor
```

For flipping the value, create it so:

```
ethabi encode function flipper.abi flip | xxd -r -p  > flip
```

You'll also need a yaml file with the following contents. Save it to flipper.yaml.

```
name: flipper
version: '1.0'
wasm: flipper.wasm
inputs:
- '12cd3c'
outputs:
- '12cd3c'
```

Now we have to start the Sawtooth Sabre environment. First clone the Sawtooth Sabre github repo and then run:

```
docker-compose -f docker-compose-installed.yaml up --build
```

Now enter the sabre-cli container:

```
docker exec -it sabre-cli bash
```

To create the flipper contract, run the following:

```
sabre cr --create flipper --owner $(cat /root/.sawtooth/keys/root.pub) --url http://
↪rest-api:9708
sabre upload --filename flipper.yaml --url http://rest-api:9708
sabre ns --create 12cd3c --url http://rest-api:9708 --owner $(cat /root/.sawtooth/
↪keys/root.pub)
sabre perm 12cd3c flipper --read --write --url http://rest-api:9708
```

To run the constructor, run:

```
sabre exec --contract flipper:1.0 --payload  ./constructor --inputs 12cd3c  --outputs␣
↪12cd3c --url http://rest-api:9708
```

Lastly, to run the flip function:

```
sabre exec --contract flipper:1.0 --payload  ./flip --inputs 12cd3c  --outputs 12cd3c␣
↪--url http://rest-api:9708
```

Warning: Returning values from Solidity is not yet implemented, and neither is `revert()`. If you attempt to call a function which returns a value, it will fail.

3.5 Using Solang with Hyperledger Burrow

In Burrow, Solang is used transparently by the `burrow deploy` tool if it is given the `--wasm` argument. When building and deploying a Solidity contract, rather than running the `solc` compiler, it will run the `solang` compiler and deploy it as a wasm contract.

This is documented in the burrow documentation.

CHAPTER 4

Solidity Language

The Solidity language supported by Solang aims to be compatible with the latest Ethereum Foundation Solidity Compiler, version 0.7 with some caveats, please check out our *Status* page.

Note: Where differences exist between different targets or the Ethereum Foundation Solidity compiler, this is noted in boxes like these.

4.1 Solidity Source File Structure

A single Solidity source file may define multiple contracts. A contract is defined with the `contract` keyword, following by the contract name and then the definition of the contract in between curly braces `{` and `}`.

```
contract A {
    /// foo simply returns true
    function foo() public returns (bool) {
        return true;
    }
}

contract B {
    /// bar simply returns false
    function bar() public returns (bool) {
        return false;
    }
}
```

When compiling this, Solang will output contract code for both *A* and *B*, irrespective of the name of source file. Although multiple contracts maybe defined in one solidity source file, it might be convenient to define a single contract in each file with the same name as the file name.

4.1.1 Imports

The `import` directive is used to import from other Solidity files. This can be useful to keep a single definition in one file, which can be used in multiple other files. Solidity imports are somewhat similar to JavaScript ES6, however there is no export statement, or default export.

The following can be imported:

- global constants
- struct definitions
- enums definitions
- event definitions
- global functions
- contracts, including abstract contract, libraries, and interfaces

There are a few different flavours of import. You can specify if you want everything imported, or a just a select few. You can also rename the imports. The following directive imports only *foo* and *bar*:

```
import {foo, bar} from "defines.sol";
```

Solang will look for the file *defines.sol* in the same directory as the current file. You can specify more directories to search with the `--importpath` commandline option. Just like with ES6, `import` is hoisted to the top and both *foo* and *bar* are usuable even before the `import` statement. It is also possible to import everything from *defines.sol* by leaving the list out. Note that this is different than ES6, which would import nothing with this syntax.

```
import "defines.sol";
```

Everything defined in *defines.sol* is now usable in your Solidity file. However, if something with the same name is defined in *defines.sol* and also in the current file, you will get a warning. Note that that it is legal to import the same file more than once.

It is also possible to rename an import. In this case, only type *foo* will be imported, and *bar* will be imported as *baz*. This is useful if you have already have a *bar* and you want to avoid a naming conflict.

```
import {bar as baz,foo} from "defines.sol";
```

Rather than renaming individual imports, it is also possible to make all the types in a file available under a special import object. In this case, the *bar* defined in *defines.sol* can is now visible as *defs.bar*, and *foo* is *defs.foo*. As long as there is no previous type *defs*, there can be no naming conflict.

```
import "defines.sol" as defs;
```

This also has a slightly more baroque syntax, which does exactly the same.

```
import * as defs from "defines.sol";
```

4.1.2 Pragmas

A pragma value is a special directive to the compiler. It has a name, and a value. The name is an identifier and the value is any text terminated by a semicolon *;*. Solang parses pragmas but does not recognise any.

Often, Solidity source files start with a `pragma solidity` which specifies the Ethereum Foundation Solidity compiler version which is permitted to compile this code. Solang does not follow the Ethereum Foundation Solidity

compiler version numbering scheme, so these pragma statements are silently ignored. There is no need for a `pragma solidity` statement when using Solang.

```
pragma solidity >=0.4.0 <0.4.8;
pragma experimental ABIEncoderV2;
```

The *ABIEncoderV2* pragma is not needed with Solang; structures can always be ABI encoded or decoded. All other pragma statements are ignored, but generate warnings.

4.2 Types

The following primitive types are supported.

4.2.1 Boolean Type

`bool` This represents a single value which can be either `true` or `false`.

4.2.2 Integer Types

`uint` This represents a single unsigned integer of 256 bits wide. Values can be for example `0`, `102`, `0xdeadcafe`, or `1000_000_000_000_000`.

`uint64`, `uint32`, `uint16`, `uint8` These represent shorter single unsigned integers of the given width. These widths are most efficient and should be used whenever possible.

`uintN` These represent shorter single unsigned integers of width `N`. `N` can be anything between 8 and 256 bits and a multiple of 8, e.g. `uint24`.

`int` This represents a single signed integer of 256 bits wide. Values can be for example `-102`, `0`, `102` or `-0xdead_cafe`.

`int64`, `int32`, `int16`, `int8` These represent shorter single signed integers of the given width. These widths are most efficient and should be used whenever possible.

`intN` These represent shorter single signed integers of width `N`. `N` can be anything between 8 and 256 bits and a multiple of 8, e.g. `int128`.

Underscores _ are allowed in numbers, as long as the number does not start with an underscore. `1_000` is allowed but `_1000` is not. Similarly `0xffff_0000` is fine, but `0x_f` is not.

Scientific notation is supported, e.g. `1e6` is one million. Only integer values are supported.

Assigning values which cannot fit into the type gives a compiler error. For example:

```
uint8 foo = 300;
```

The largest value an `uint8` can hold is $(2^8) - 1 = 255$. So, the compiler says:

```
implicit conversion would truncate from uint16 to uint8
```

Tip: When using integers, whenever possible use the `int64`, `int32` or `uint64`, `uint32` types.

The Solidity language has its origins for the Ethereum Virtual Machine (EVM), which has support for 256 bit arithmetic. Most common CPUs like x86_64 do not implement arithmetic for such large types, and any EVM virtual machine implementation has to do bigint calculations, which are expensive.

WebAssembly or BPF do not support this. As a result that Solang has to emulate larger types with many instructions, resulting in larger contract code and higher gas cost.

4.2.3 Fixed Length byte arrays

Solidity has a primitive type unique to the language. It is a fixed-length byte array of 1 to 32 bytes, declared with *bytes* followed by the array length, for example: `bytes32`, `bytes24`, `bytes8`, or `bytes1`. `byte` is an alias for `byte1`, so `byte` is an array of 1 element. The arrays can be initialized with either a hex string or a text string.

```
bytes4 foo = "ABCD";
bytes4 bar = hex"41_42_43_44";
```

The ascii value for `A` is 41 in hexadecimal. So, in this case, foo and bar are initialized to the same value. Underscores are allowed in hex strings; they exist for readability. If the string is shorter than the type, it is padded with zeros. For example:

```
bytes6 foo = "AB" "CD";
bytes5 bar = hex"41";
```

String literals can be concatenated like they can in C or C++. Here the types are longer than the initializers; this means they are padded at the end with zeros. foo will contain the following bytes in hexadecimal `41 42 43 44 00 00` and bar will be `41 00 00 00 00`.

These types can be used with all the bitwise operators, `~`, `|`, `&`, `^`, `<<`, and `>>`. When these operators are used, the type behaves like an unsigned integer type. In this case think the type not as an array but as a long number. For example, it is possible to shift by one bit:

```
bytes2 foo = hex"0101" << 1;
// foo is 02 02
```

Since this is an array type, it is possible to read array elements too. They are indexed from zero. It is not permitted to set array elements; the value of a bytesN type can only be changed by setting the entire array value.

```
bytes6 wake_code = "heotymeo";
bytes1 second_letter = wake_code[1]; // second_letter is "e"
```

The length can be read using the `.length` member variable. Since this is a fixed size array, this is always the length of the type itself.

```
bytes32 hash;
assert(hash.length == 32);
byte b;
assert(b.length == 1);
```

4.2.4 Address and Address Payable Type

The `address` type holds the address of an account. The length of an `address` type depends on the target being compiled for. On ewasm, an address is 20 bytes. Substrate has an address length of 32 bytes. The format of an address literal depends on what target you are building for. On ewasm, ethereum addresses can be specified with a particular hexadecimal number.

```
address foo = 0xE9430d8C01C4E4Bb33E44fd7748942085D82fC91;
```

The hexadecimal string has to have 40 characters, and not contain any underscores. The capitalization, i.e. whether `a` to `f` values are capitalized, is important. It is defined in EIP-55. For example, when compiling:

```
address foo = 0xe9430d8C01C4E4Bb33E44fd7748942085D82fC91;
```

Since the hexadecimal string is 40 characters without underscores, and the string does not match the EIP-55 encoding, the compiler will refused to compile this. To make this a regular hexadecimal number, not an address literal, add some leading zeros or some underscores. In order to fix the address literal, copy the address literal from the compiler error message:

```
error: address literal has incorrect checksum, expected␣
↪'0xE9430d8C01C4E4Bb33E44fd7748942085D82fC91'
```

Substrate or Solana addresses are base58 encoded, not hexadecimal. An address literal can be specified with the special syntax `address"<account>"`.

```
address foo = address"5GBWmgdFAMqm8ZgAHGobqDqX6tjLxJhv53ygjNtaaAn3sjeZ";
```

An address can be payable or not. An payable address can used with the *.send() and .transfer() functions*, and *selfdestruct(address payable recipient)* function. A non-payable address or contract can be cast to an `address payable` using the `payable()` cast, like so:

```
address payable addr = payable(this);
```

`address` cannot be used in any arithmetic or bitwise operations. However, it can be cast to and from bytes types and integer types. The == and `!=` operators work for comparing two address types.

```
address foo = address(0);
```

Note: The type name `address payable` cannot be used as a cast in the Ethereum Foundation Solidity compiler, and the cast must be `payable` instead. This is apparently due to a limitation in their parser. Solang's generated parser has no such limitation and allows `address payable` to be used as a cast, but allows `payable` to be used as a cast well, for compatibility reasons.

Note: Substrate can be compiled with a different type for Address. If you need support for a different length than the default, please get in touch.

4.2.5 Enums

Solidity enums types need to have a definition which lists the possible values it can hold. An enum has a type name, and a list of unique values. Enum types can used in public functions, but the value is represented as a `uint8` in the ABI.

```
contract enum_example {
    enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }

    function is_weekend(Weekday day) public pure returns (bool) {
        return (day == Weekday.Saturday || day == Weekday.Sunday);
    }
}
```

An enum can be converted to and from integer, but this requires an explicit cast. The value of an enum is numbered from 0, like in C and Rust.

If enum is declared in another contract, the type can be refered to with *contractname.typename*. The individual enum values are *contractname.typename.value*. The enum declaration does not have to appear in a contract, in which case it can be used without the contract name prefix.

```
enum planets { Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune }

contract timeofday {
    enum time { Night, Day, Dawn, Dusk }
}

contract stargazing {
    function look_for(timeofday.time when) public returns (planets[]) {
        if (when == timeofday.time.Dawn || when == timeofday.time.Dusk) {
            planets[] x = new planets[](2);
            x[0] = planets.Mercury;
            x[1] = planets.Venus;
            return x;
        } else if (when == timeofday.time.Night) {
            planets[] x = new planets[](5);
            x[0] = planets.Mars;
            x[1] = planets.Jupiter;
            x[2] = planets.Saturn;
            x[3] = planets.Uranus;
            x[4] = planets.Neptune;
            return x;
        } else {
            planets[] x = new planets[](1);
            x[0] = planets.Earth;
            return x;
        }
    }
}
```

4.2.6 Struct Type

A struct is composite type of several other types. This is used to group related items together.

```
contract deck {
    enum suit { club, diamonds, hearts, spades }
    enum value { two, three, four, five, six, seven, eight, nine, ten, jack, queen,␣
↪king, ace }
    struct card {
        value v;
        suit s;
    }

    function score(card c) public returns (uint32 score) {
        if (c.s == suit.hearts) {
            if (c.v == value.ace) {
                score = 14;
            }
            if (c.v == value.king) {
                score = 13;
```

(continues on next page)

(continued from previous page)

```
            }
            if (c.v == value.queen) {
                score = 12;
            }
            if (c.v == value.jack) {
                score = 11;
            }
        }
        // all others score 0
    }
}
```

A struct has one or more fields, each with a unique name. Structs can be function arguments and return values. Structs can contain other structs. There is a struct literal syntax to create a struct with all the fields set.

```
contract deck {
    enum suit { club, diamonds, hearts, spades }
    enum value { two, three, four, five, six, seven, eight, nine, ten, jack, queen,␣
↪king, ace }
    struct card {
        value v;
        suit s;
    }

    card card1 = card(value.two, suit.club);
    card card2 = card({s: suit.club, v: value.two});

    // This function does a lot of copying
    function set_card1(card c) public returns (card previous) {
        previous = card1;
        card1 = c;
    }
}
```

The two contract storage variables `card1` and `card2` have initializers using struct literals. Struct literals can either set fields by their position, or field name. In either syntax, all the fields must be specified. When specifying structs fields by position, the order of the fields must match with the struct definition. When fields are specified by name, the order is not important.

Struct definitions from other contracts can be used, by referring to them with the *contractname.* prefix. Struct definitions can appear outside of contract definitions, in which case they can be used in any contract without the prefix.

```
struct user {
    string name;
    bool active;
}

contract auth {
    function authenticate(string name, db.users storage users) public returns (bool) {
        // ...
    }
}

contract db {
    struct users {
        user[] field1;
```

(continues on next page)

(continued from previous page)

```
        int32 count;
    }
}
```

The *users* struct contains an array of *user*, which is another struct. The *users* struct is defined in contract *db*, and can be used in another contract with the type name *db.users*. Astute readers may have noticed that the *db.users* struct is used before it is declared. In Solidity, types can be always be used before their declaration, or before they are imported.

Structs can be contract storage variables. Structs in contract storage can be assigned to structs in memory and vice versa, like in the *set_card1()* function. Copying structs between storage and memory is expensive; code has to be generated for each field and executed.

- The function argument `c` has to ABI decoded (1 copy + decoding overhead)
- The `card1` has to load from contract storage (1 copy + contract storage overhead)
- The `c` has to be stored into contract storage (1 copy + contract storage overhead)
- The `previous` struct has to ABI encoded (1 copy + encoding overhead)

Note that struct variables are references. When contract struct variables or normal struct variables are passed around, just the memory address or storage slot is passed around internally. This makes it very cheap, but it does mean that if a called function modifies the struct, then this is visible in the caller as well.

```
contract foo {
    struct bar {
        bytes32 f1;
        bytes32 f2;
        bytes32 f3;
        bytes32 f4;
    }

    function f(bar b) public {
        b.f4 = "foobar";
    }

    function example() public {
        bar bar1;

        // bar1 is passed by reference; just its pointer is passed
        f(bar1);

        assert(bar1.f4 == "foobar");
    }
}
```

Note: In the Ethereum Foundation Solidity compiler, you need to add `pragma experimental ABIEncoderV2;` to use structs as return values or function arguments in public functions. The default ABI encoder of Solang can handle structs, so there is no need for this pragma. The Solang compiler ignores this pragma if present.

4.2.7 Fixed Length Arrays

Arrays can be declared by adding [length] to the type name, where length is a constant expression. Any type can be made into an array, including arrays themselves (also known as arrays of arrays). For example:

```
contract foo {
    /// In a vote with 11 voters, do the ayes have it?
    function f(bool[11] votes) public pure returns (bool) {
        uint32 i;
        uint32 ayes = 0;

        for (i=0; i<votes.length; i++) {
            if (votes[i]) {
                ayes += 1;
            }
        }

        // votes.length is odd; integer truncation means that 11 / 2 = 5
        return ayes > votes.length / 2;
    }
}
```

Note the length of the array can be read with the `.length` member. The length is readonly. Arrays can be initialized with an array literal. The first element of the array should be cast to the correct element type. For example:

```
contract primes {
    function primenumber(uint32 n) public pure returns (uint64) {
        uint64[10] primes = [ uint64(2), 3, 5, 7, 11, 13, 17, 19, 23, 29 ];

        return primes[n];
    }
}
```

Any array subscript which is out of bounds (either an negative array index, or an index past the last element) will cause a runtime exception. In this example, calling `primenumber(10)` will fail; the first prime number is indexed by 0, and the last by 9.

Arrays are passed by reference. If you modify the array in another function, those changes will be reflected in the current function. For example:

```
contract reference {
    function set_2(int8[4] a) pure private {
        a[2] = 102;
    }

    function foo() private {
        int8[4] val = [ int8(1), 2, 3, 4 ];

        set_2(val);

        // val was passed by reference, so was modified
        assert(val[2] == 102);
    }
}
```

Note: In Solidity, an fixed array of 32 bytes (or smaller) can be declared as `bytes32` or `uint8[32]`. In the Ethereum ABI encoding, an `int8[32]` is encoded using 32 × 32 = 1024 bytes. This is because the Ethereum ABI encoding pads each primitive to 32 bytes. However, since `bytes32` is a primitive in itself, this will only be 32 bytes when ABI encoded.

In Substrate, the SCALE encoding uses 32 bytes for both types.

4.2.8 Dynamic Length Arrays

Dynamic length arrays are useful for when you do not know in advance how long your arrays will need to be. They are declared by adding `[]` to your type. How they can be used depends on whether they are contract storage variables or stored in memory.

Memory dynamic arrays must be allocated with `new` before they can be used. The `new` expression requires a single unsigned integer argument. The length can be read using `length` member variable. Once created, the length of the array cannot be changed.

```
contract dynamicarray {
    function test(uint32 size) public {
        int64[] memory a = new int64[](size);

        for (uint32 i = 0; i < size; i++) {
            a[i] = 1 << i;
        }

        assert(a.length == size);
    }
}
```

Note: There is experimental support for *push()* and *pop()* on memory arrays.

Storage dynamic memory arrays do not have to be allocated. By default, they have a length of zero and elements can be added and removed using the `push()` and `pop()` methods.

```
contract s {
    int64[] a;

    function test() public {
        // push takes a single argument with the item to be added
        a.push(128);
        // push with no arguments adds 0
        a.push();
        // now we have two elements in our array, 128 and 0
        assert(a.length == 2);
        a[0] |= 64;
        // pop removes the last element
        a.pop();
        // you can assign the return value of pop
        int64 v = a.pop();
        assert(v == 192);
    }
}
```

Calling the method `pop()` on an empty array is an error and contract execution will abort, just like when you access an element beyond the end of an array.

`push()` without any arguments returns a storage reference. This is only available for types that support storage references (see below).

```
contract example {
    struct user {
        address who;
        uint32 hitcount;
```

(continues on next page)

(continued from previous page)

```
    }
    s[] foo;

    function test() public {
        // foo.push() creates an empty entry and returns a reference to it
        user storage x = foo.push();

        x.who = address(1);
        x.hitcount = 1;
    }
}
```

Depending on the array element, `pop()` can be costly. It has to first copy the element to memory, and then clear storage.

4.2.9 String

Strings can be initialized with a string literal or a hex literal. Strings can be concatenated and compared, and formatted using *.format()*; no other operations are allowed on strings.

```
contract example {
    function test1(string s) public returns (bool) {
        string str = "Hello, " + s + "!";

        return (str == "Hello, World!");
    }

    function test2(string s, int64 n) public returns (string res) {
        res = "Hello, {}! #{}".format(s, n);
    }
}
```

Strings can be cast to *bytes*. This cast has no runtime cost, since both types use the same underlying data structure.

Note: The Ethereum Foundation Solidity compiler does not allow unicode characters in string literals, unless it is prefixed with unicode, e.g. `unicode"€"` . For compatibility, Solang also accepts the unicode prefix. Solang always allows unicode characters in strings.

4.2.10 Dynamic Length Bytes

The `bytes` datatype is a dynamic length array of bytes. It can be created with the `new` operator, or from an string or hex initializer. Unlike the `string` type, it is possible to index the `bytes` datatype like an array.

```
contract b {
    function test() public {
        bytes a = hex"0000_00fa";
        bytes b = new bytes(4);

        b[3] = hex"fa";

        assert(a == b);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If the `bytes` variable is a storage variable, there is a `push()` and `pop()` method available to add and remove bytes from the array. Array elements in a memory `bytes` can be modified, but no elements can be removed or added, in other words, `push()` and `pop()` are not available when `bytes` is stored in memory.

A `string` type can be cast to `bytes`. This way, the string can be modified or characters can be read. Note this will access the string by byte, not character, so any non-ascii characters will need special handling.

An dynamic array of bytes can use the type `bytes` or `byte[]`. The latter stores each byte in an individual storage slot, while the former stores the entire string in a single storage slot, when possible. Additionally a `string` can be cast to `bytes` but not to `byte[]`.

4.2.11 Mappings

Mappings are a dictionary type, or associative arrays. Mappings have a number of limitations:

- it has to be in contract storage, not memory
- they are not iterable
- the key cannot be a `struct`, array, or another mapping.

Mappings are declared with `mapping(keytype => valuetype)`, for example:

```
contract b {
    struct user {
        bool exists;
        address addr;
    }
    mapping(string => user) users;

    function add(string name, address addr) public {
        // assigning to a storage variable creates a reference
        user storage s = users[name];

        s.exists = true;
        s.addr = addr;
    }

    function get(string name) public view returns (bool, address) {
        // assigning to a memory variable creates a copy
        user s = users[name];

        return (s.exists, s.addr);
    }

    function rm(string name) public {
        delete users[name];
    }
}
```

Tip: When assigning multiple members in a struct in a mapping, it is better to create a storage variable as a reference to the struct, and then assign to the reference. The add() function above could have been written as:

```
function add(string name, address addr) public {
    s[name].exists = true;
    s[name].addr = addr;
}
```

Here the storage slot for struct is calculated twice, which includes an expensive keccak256 calculation.

If you access a non-existing field on a mapping, all the fields will read as zero. So, it is common practise to have a boolean field called `exists`. Since mappings are not iterable, it is not possible to do a `delete` on an mapping, but an entry can be deleted.

Note: Solidity takes the keccak 256 hash of the key and the storage slot, and simply uses that to find the entry. There are no hash collision chains. This scheme is simple and avoids "hash flooding" attacks where the attacker chooses data which hashes to the same hash collision chain, making the hash table very slow; it will behave like a linked list.

In order to implement mappings in memory, a new scheme must be found which avoids this attack. Usually this is done with SipHash, but this cannot be used in smart contracts since there is no place to store secrets. Collision chains are needed since memory has a much smaller address space than the 256 bit storage slots.

Any suggestions for solving this are very welcome!

4.2.12 Contract Types

In Solidity, other smart contracts can be called and created. So, there is a type to hold the address of a contract. This is in fact simply the address of the contract, with some syntax sugar for calling functions it.

A contract can be created with the new statement, followed by the name of the contract. The arguments to the constructor must be provided.

```
contract child {
    function announce() public {
        print("Greetings from child contract");
    }
}

contract creator {
    function test() public {
        child c = new child();

        c.announce();
    }
}
```

Since child does not have a constructor, no arguments are needed for the new statement. The variable *c* of the contract *child* type, which simply holds its address. Functions can be called on this type. The contract type can be cast to and from address, provided an explicit cast is used.

The expression `this` evaluates to the current contract, which can be cast to `address` or `address payable`.

```
contract example {
    function get_address() public returns (address) {
        return address(this);
    }
}
```

4.2.13 Function Types

Function types are references to functions. You can use function types to pass functions for callbacks, for example. Function types come in two flavours, `internal` and `external`. An internal function is a reference to a function in the same contract or one of its base contracts. An external function is a reference to a public or external function on any contract.

When declaring a function type, you must specify the parameters types, return types, mutability, and whether it is external or internal. The parameters or return types cannot have names.

```
contract ft {
    function test() public {
        // reference to an internal function with two argments, returning bool
        // with the default mutability (i.e. cannot be payable)
        function(int32, bool) internal returns (bool) x;

        // the local function func1 can be assigned to this type; mutability
        // can be more restrictive than the type.
        x = func1;

        // now you can call func1 via the x
        bool res = x(102, false);

        // reference to an internal function with no return values, must be pure
        function(int32 arg1, bool arg2) internal pure y;

        // Does not compile: wrong number of return types and mutability
        // is not compatible.
        y = func1;
    }

    function func1(int32 arg, bool arg2) view internal returns (bool) {
        return false;
    }
}
```

If the `internal` or `external` keyword is omitted, the type defaults to internal.

Just like any other type, a function type can be a function argument, function return type, or a contract storage variable. Internal function types cannot be used in public functions parameters or return types.

An external function type is a reference to a function in a particular contract. It stores the address of the contract, and the function selector. An internal function type only stores the function reference. When assigning a value to an external function selector, the contract and function must be specified, by using a function on particular contract instance.

```
contract ft {
    function test(paffling p) public {
        // this.callback can be used as an external function type value
        p.set_callback(this.callback);
    }

    function callback(int32 count, string foo) public {
        // ...
    }
}

contract paffling {
```

(continues on next page)

(continued from previous page)

```
    // the first visibility "external" is for the function type, the second "internal
↪" is
    // for the callback variables
    function(int32, string) external internal callback;

    function set_callback(function(int32, string) external c) public {
        callback = c;
    }

    function piffle() public {
        callback(1, "paffled");
    }
}
```

4.2.14 Storage References

Parameters, return types, and variables can be declared storage references by adding `storage` after the type name. This means that the variable holds a references to a particular contract storage variable.

```
contract felix {
    enum Felines { None, Lynx, Felis, Puma, Catopuma };
    Felines[100] group_a;
    Felines[100] group_b;


    function count_pumas(Felines[100] storage cats) private returns (uint32)
{
        uint32 count = 0;
        uint32 i = 0;

        for (i = 0; i < cats.length; i++) {
            if (cats[i] == Felines.Puma) {
                ++count;
            }
        }

        return count;
    }

    function all_pumas() public returns (uint32) {
        Felines[100] storage ref = group_a;

        uint32 total = count_pumas(ref);

        ref = group_b;

        total += count_pumas(ref);

        return total;
    }
}
```

Functions which have either storage parameter or return types cannot be public; when a function is called via the ABI encoder/decoder, it is not possible to pass references, just values. However it is possible to use storage reference variables in public functions, as demonstrated in function all_pumas().

4.3 Expressions

Solidity resembles the C family of languages. Expressions can use the following operators.

4.3.1 Arithmetic operators

The binary operators `-`, `+`, `*`, `/`, `%`, and `**` are supported, and also in the assignment form `-=`, `+=`, `*=`, `/=`, and `%=`. There is a unary operator `-`.

```
uint32 fahrenheit = celcius * 9 / 5 + 32;
```

Parentheses can be used too, of course:

```
uint32 celcius = (fahrenheit - 32) * 5 / 9;
```

Operators can also come in the assignment form.

```
balance += 10;
```

The exponation (or power) can be used to multiply a number N times by itself, i.e. x^y. This can only be done for unsigned types.

```
uint64 thousand = 1000;
uint64 billion = thousand ** 3;
```

Note: No overflow checking is done on the arithmetic operations, just like with the Ethereum Foundation Solidity compiler.

4.3.2 Bitwise operators

The `|`, `&`, `^` are supported, as are the shift operators << and >>. These are also available in the assignment form `|=`, `&=`, `^=`, <<=, and >>=. Lastly there is a unary operator `~` to invert all the bits in a value.

4.3.3 Logical operators

The logical operators `||`, `&&`, and `!` are supported. The `||` and `&&` short-circuit. For example:

```
bool foo = x > 0 || bar();
```

bar() will not be called if the left hand expression evaluates to true, i.e. x is greater than 0. If x is 0, then bar() will be called and the result of the `||` will be the return value of bar(). Similarly, the right hand expressions of `&&` will not be evaluated if the left hand expression evaluates to `false`; in this case, whatever ever the outcome of the right hand expression, the `&&` will result in `false`.

```
bool foo = x > 0 && bar();
```

Now `bar()` will only be called if x *is* greater than 0. If x is 0 then the `&&` will result in false, irrespective of what bar() would return, so bar() is not called at all. The expression elides execution of the right hand side, which is also called *short-circuit*.

4.3.4 Conditional operator

The ternary conditional operator `? :` is supported:

```
uint64 abs = foo > 0 ? foo : -foo;
```

4.3.5 Comparison operators

It is also possible to compare values. For, this the >=, >, ==, !=, <, and <= is supported. This is useful for conditionals.

The result of a comparison operator can be assigned to a bool. For example:

```
bool even = (value % 2) == 0;
```

It is not allowed to assign an integer to a bool; an explicit comparision is needed to turn it into a bool.

4.3.6 Increment and Decrement operators

The post-increment and pre-increment operators are implemented like you would expect. So, `a++` evaluates to the value of `a` before incrementing, and `++a` evaluates to value of `a` after incrementing.

4.3.7 this

The keyword `this` evaluates to the current contract. The type of this is the type of the current contract. It can be cast to `address` or `address payable` using a cast.

```
contract kadowari {
    function nomi() public {
        kadowari c = this;
        address a = address(this);
    }
}
```

Function calls made via this are function calls through the external call mechanism; i.e. they have to serialize and deserialise the arguments and have the external call overhead. In addition, this only works with public functions.

```
contract kadowari {
    function nomi() public {
        this.nokogiri(102);
    }

    function nokogiri(int a) public {
        // ...
    }
}
```

4.3.8 type(..) operators

For integer values, the minimum and maximum values the types can hold are available using the `type(...).min` and `type(...).max` operators. For unsigned integers, `type(..).min` will always be 0.

```
contract example {
    int16 stored;

    function func(int x) public {
        if (x < type(int16).min || x > type(int16).max) {
            revert("value will not fit");
        }

        stored = int16(x);
    }
}
```

The contract code for a contract, i.e. the binary WebAssembly or BOF, can be retrieved using the `type(c).creationCode` and `type(c).runtimeCode` fields, as `bytes`. In Ethereum, the constructor code is in the `creationCode` WebAssembly and all the functions are in the `runtimeCode` WebAssembly or BPF. Parity Substrate has a single WebAssembly code for both, so both fields will evaluate to the same value.

```
contract example {
    function test() public {
        bytes runtime = type(other).runtimeCode;
    }
}

contract other {
    bool foo;
}
```

Note: `type().creationCode` and `type().runtimeCode` are compile time constants.

It is not possible to access the code for the current contract. If this were possible, then the contract code would need to contain itself as a constant array, which would result in an contract of infinite size.

4.3.9 Ether and time units

Any decimal numeric literal constant can have a unit denomination. For example `10 minutes` will evaluate to 600, i.e. the constant will be multiplied by the multiplier listed below. The following units are available:

Unit	Multiplier
`seconds`	1
`minutes`	60
`hours`	3600
`days`	86400
`weeks`	604800
`wei`	1
`szabo`	1_000_000_000_000
`finney`	1_000_000_000_000_000
`ether`	1_000_000_000_000_000_000

Note that `ether`, `wei` and the other Ethereum currency denominations are available when not compiling for Ethereum, but they will produce warnings.

4.3.10 Casting

Solidity is very strict about the sign of operations, and whether an assignment can truncate a value. You can force the compiler to accept truncations or differences in sign by adding a cast.

Some examples:

```
function abs(int bar) public returns (int64) {
    if (bar > 0) {
        return bar;
    } else {
        return -bar;
    }
}
```

The compiler will say:

```
implicit conversion would truncate from int256 to int64
```

Now you can work around this by adding a cast to the argument to return `return int64(bar);`, however it would be much nicer if the return value matched the argument. Instead, implement multiple overloaded abs() functions, so that there is an `abs()` for each type.

It is allowed to cast from a `bytes` type to `int` or `uint` (or vice versa), only if the length of the type is the same. This requires an explicit cast.

```
bytes4 selector = "ABCD";
uint32 selector_as_uint = uint32(selector);
```

If the length also needs to change, then another cast is needed to adjust the length. Truncation and extension is different for integers and bytes types. Integers pad zeros on the left when extending, and truncate on the right. bytes pad on right when extending, and truncate on the left. For example:

```
bytes4 start = "ABCD";
uint64 start1 = uint64(uint4(start));
// first cast to int, then extend as int: start1 = 0x41424344
uint64 start2 = uint64(bytes8(start));
// first extend as bytes, then cast to int: start2 = 0x4142434400000000
```

A similar example for truncation:

```
uint64 start = 0xdead_cafe;
bytes4 start1 = bytes4(uint32(start));
// first truncate as int, then cast: start1 = hex"cafe"
bytes4 start2 = bytes4(bytes8(start));
// first cast, then truncate as bytes: start2 = hex"dead"
```

Since `byte` is array of one byte, a conversion from `byte` to `uint8` requires a cast.

4.4 Statements

In functions, you can declare variables in code blocks. If the name is the same as an existing function, enum type, or another variable, then the compiler will generate a warning as the original item is no longer accessible.

```
contract test {
    uint foo = 102;
    uint bar;

    function foobar() private {
        // AVOID: this shadows the contract storage variable foo
        uint foo = 5;
    }
}
```

Scoping rules apply as you would expect, so if you declare a variable in a block, then it is not accessible outside that block. For example:

```
 function foo() public {
    // new block is introduced with { and ends with }
    {
        uint a;

        a = 102;
    }

    // ERROR: a is out of scope
    uint b = a + 5;
}
```

4.4.1 If statement

Conditional execution of a block can be achieved using an `if (condition) { }` statement. The condition must evaluate to a `bool` value.

```
function foo(uint32 n) private {
    if (n > 10) {
        // do something
    }

    // ERROR: unlike C integers can not be used as a condition
    if (n) {
          // ...
    }
}
```

The statements enclosed by `{` and `}` (commonly known as a *block*) are executed only if the condition evaluates to true.

4.4.2 While statement

Repeated execution of a block can be achieved using `while`. It syntax is similar to `if`, however the block is repeatedly executed until the condition evaluates to false. If the condition is not true on first execution, then the loop is never executed:

```
function foo(uint n) private {
    while (n >= 10) {
        n -= 9;
    }
}
```

It is possible to terminate execution of the while statement by using the `break` statement. Execution will continue to next statement in the function. Alternatively, `continue` will cease execution of the block, but repeat the loop if the condition still holds:

```
function foo(uint n) private {
    while (n >= 10) {
        n--;

        if (n >= 100) {
            // do not execute the if statement below, but loop again
            continue;
        }

        if (bar(n)) {
            // cease execution of this while loop and jump to the "n = 102" statement
            break;
        }
    }

    n = 102;
}
```

4.4.3 Do While statement

A `do { ... } while (condition);` statement is much like the `while (condition) { ... }` except that the condition is evaluated after execution the block. This means that the block is executed at least once, which is not true for `while` statements:

```
function foo(uint n) private {
    do {
        n--;

        if (n >= 100) {
            // do not execute the if statement below, but loop again
            continue;
        }

        if (bar(n)) {
            // cease execution of this while loop and jump to the "n = 102" statement
            break;
        }
    }
    while (n > 10);

    n = 102;
}
```

4.4.4 For statements

For loops are like `while` loops with added syntaxic sugar. To execute a loop, we often need to declare a loop variable, set its initial variable, have a loop condition, and then adjust the loop variable for the next loop iteration.

For example, to loop from 0 to 1000 by steps of 100:

```
function foo() private {
    for (uint i = 0; i <= 1000; i += 100) {
        // ...
    }
}
```

The declaration `uint i = 0` can be omitted if no new variable needs to be declared, and similarly the post increment `i += 100` can be omitted if not necessary. The loop condition must evaluate to a boolean, or it can be omitted completely. If it is ommited the block must contain a `break` or `return` statement, else execution will repeat infinitely (or until all gas is spent):

```
function foo(uint n) private {
    // all three omitted
    for (;;) {
        // there must be a way out
        if (n == 0) {
            break;
        }
    }
}
```

4.4.5 Destructuring Statement

The destructuring statement can be used for making function calls to functions that have multiple return values. The list can contain either:

1. The name of an existing variable. The type must match the type of the return value.
2. A new variable declaration with a type. Again, the type must match the type of the return value.
3. Empty; this return value is ignored and not accessible.

```
contract destructure {
    function func() internal returns (bool, int32, string) {
        return (true, 5, "abcd")
    }

    function test() public {
        string s;
        (bool b, , s) = func();
    }
}
```

The right hand side may also be a list of expressions. This type can be useful for swapping values, for example.

```
function test() public {
    (int32 a, int32 b, int32 c) = (1, 2, 3);

    (b, , a) = (a, 5, b);
}
```

4.4.6 Try Catch Statement

Sometimes execution gets reverted due to a `revert()` or `require()`. These types of problems usually cause the entire transaction to be aborted. However, it is possible to catch some of these problems and continue execution.

This is only possible for contract instantiation through new, and external function calls. An internal function cannot be called from a try catch statement. Not all problems can be handled, for example, out of gas cannot be caught. The `revert()` and `require()` builtins may be passed a reason code, which can be inspected using the `catch Error(string)` syntax.

```
contract aborting {
    constructor() {
        revert("bar");
    }
}

contract runner {
    function test() public {
        try new aborting() returns (aborting a) {
            // new succeeded; a holds the a reference to the new contract
        }
        catch Error(string x) {
            if (x == "bar") {
                // "bar" revert or require was executed
            }
        }
        catch (bytes raw) {
            // if no error string could decoding, we end up here with the raw data
        }
    }
}
```

The same statement can be used for calling external functions. The `returns (...)` part must match the return types for the function. If no name is provided, that return value is not accessible.

```
contract aborting {
    function abort() public returns (int32, bool) {
        revert("bar");
    }
}

contract runner {
    function test() public {
        aborting abort = new aborting();

        try abort.abort() returns (int32 a, bool b) {
            // call succeeded; return values are in a and b
        }
        catch Error(string x) {
            if (x == "bar") {
                // "bar" reason code was provided through revert() or require()
            }
        }
        catch (bytes raw) {
            // if no error string could decoding, we end up here with the raw data
        }
    }
}
```

There is an alternate syntax which avoids the abi decoding by leaving the *catch Error(...)* out. This might be useful when no error string is expected, and will generate shorter code.

```
contract aborting {
    function abort() public returns (int32, bool) {
        revert("bar");
    }
}

contract runner {
    function test() public {
        aborting abort = new aborting();

        try new abort.abort() returns (int32 a, bool b) {
            // call succeeded; return values are in a and b
        }
        catch (bytes raw) {
            // call failed with raw error in raw
        }
    }
}
```

4.5 Functions

A function can be declared inside a contract, in which case it has access to the contracts contract storage variables, other contract functions etc. Functions can be also be declared outside a contract.

```
/// get_initial_bound is called from the constructor
function get_initial_bound() returns (uint value) {
    value = 102;
}

contact foo {
    uint bound = get_initial_bound();

    /** set bound for get with bound */
    function set_bound(uint _bound) public {
        bound = _bound;
    }

    /// Clamp a value within a bound.
    /// The bound can be set with set_bound().
    function get_with_bound(uint value) view public return (uint) {
        if (value < bound) {
            return value;
        } else {
            return bound;
        }
    }
}
```

Function can have any number of arguments. Function arguments may have names; if they do not have names then they cannot be used in the function body, but they will be present in the public interface.

The return values may have names as demonstrated in the get_initial_bound() function. When at all of the return values have a name, then the return statement is no longer required at the end of a function body. In stead of returning the values which are provided in the return statement, the values of the return variables at the end of the function is returned. It is still possible to explicitly return some values with a return statement.

Functions which are declared `public` will be present in the ABI and are callable externally. If a function is declared `private` then it is not callable externally, but it can be called from within the contract. If a function is defined outside a contract, then it cannot have a visibility specifier (e.g. `public`).

Any DocComment before a function will be include in the ABI. Currently only Substrate supports documentation in the ABI.

4.5.1 Arguments passing and return values

Function arguments can be passed either by position or by name. When they are called by name, arguments can be in any order. However, functions with anonymous arguments (arguments without name) cannot be called this way.

```
contract foo {
    function bar(uint32 x, bool y) public returns (uint32) {
        if (y) {
            return 2;
        }

        return 3;
    }

    function test() public {
        uint32 a = bar(102, false);
        a = bar({ y: true, x: 302 });
    }
}
```

If the function has a single return value, this can be assigned to a variable. If the function has multiple return values, these can be assigned using the *Destructuring Statement* assignment statement:

```
contract foo {
    function bar1(uint32 x, bool y) public returns (address, byte32) {
        return (address(3), hex"01020304");
    }

    function bar2(uint32 x, bool y) public returns (bool) {
        return !y;
    }

    function test() public {
        (address f1, bytes32 f2) = bar1(102, false);
        bool f3 = bar2({x: 255, y: true})
    }
}
```

It is also possible to call functions on other contracts, which is also known as calling external functions. The called function must be declared public. Calling external functions requires ABI encoding the arguments, and ABI decoding the return values. This much more costly than an internal function call.

```
contract foo {
    function bar1(uint32 x, bool y) public returns (address, byte32) {
        return (address(3), hex"01020304");
    }

    function bar2(uint32 x, bool y) public returns (bool) {
        return !y;
```

(continues on next page)

(continued from previous page)

```
    }
}

contract bar {
    function test(foo f) public {
        (address f1, bytes32 f2) = f.bar1(102, false);
        bool f3 = f.bar2({x: 255, y: true})
    }
}
```

The syntax for calling external call is the same as the external call, except for that it must be done on a contract type variable. Any error in an external call can be handled with *Try Catch Statement*.

4.5.2 Passing value and gas with external calls

For external calls, value can be sent along with the call. The callee must be `payable`. Likewise, a gas limit can be set.

```
contract foo {
    function bar() public {
        other o = new other();

        o.feh{value: 102, gas: 5000}(102);
    }
}

contract other {
    function feh(uint32 x) public payable {
        // ...
    }
}
```

4.5.3 State mutability

Some functions only read contract storage (also known as *state*), and others may write contract storage. Functions that do not write state can be executed off-chain. Off-chain execution is faster, does not require write access, and does not need any balance.

Functions that do not write state come in two flavours: `view` and `pure`. `pure` functions may not read state, and `view` functions that do read state.

Functions that do write state come in two flavours: `payable` and non-payable, the default. Functions that are not intended to receive any value, should not be marked `payable`. The compiler will check that every call does not included any value, and there are runtime checks as well, which cause the function to be reverted if value is sent.

A constructor can be marked `payable`, in which case value can be passed with the constructor.

Note: If value is sent to a non-payable function on Parity Substrate, the call will be reverted. However there is no refund performed, so value will remain with the callee.

`payable` on constructors is not enforced on Parity Substrate. Funds are needed for storage rent and there is a minimum deposit needed for the contract. As a result, constructors always receive value on Parity Substrate.

4.5.4 Function overloading

Multiple functions with the same name can be declared, as long as the arguments are different in at least one of two ways:

- The number of arguments must be different
- The type of at least one of the arguments is different

A function cannot be overloaded by changing the return types or number of returned values. Here is an example of an overloaded function:

```
contract shape {
    int64 bar;

    function abs(int val) public returns (int) {
        if (val >= 0) {
            return val;
        } else {
            return -val;
        }
    }

    function abs(int64 val) public returns (int64) {
        if (val >= 0) {
            return val;
        } else {
            return -val;
        }
    }

    function foo(int64 x) public {
        bar = abs(x);
    }
}
```

In the function foo, abs() is called with an `int64` so the second implementation of the function abs() is called.

4.5.5 Function Modifiers

Function modifiers are used to check pre-conditions or post-conditions for a function call. First a new modifier must be declared which looks much like a function, but uses the `modifier` keyword rather than `function`.

```
contract example {
    address owner;

    modifier only_owner() {
        require(msg.sender == owner);
        _;
        // insert post conditions here
    }

    function foo() only_owner public {
        // ...
    }
}
```

The function *foo* can only be run by the owner of the contract, else the `require()` in its modifier will fail. The special symbol `_;` will be replaced by body of the function. In fact, if you specify `_;` twice, the function will execute twice, which might not be a good idea.

A modifier cannot have visibility (e.g. `public`) or mutability (e.g. `view`) specified, since a modifier is never externally callable. Modifiers can only be used by attaching them to functions.

A modifier can have arguments, just like regular functions. Here if the price is less than 50, *foo()* itself will never be executed, and execution will return to the caller with nothing done since `_;` is not reached in the modifier and as result foo() is never executed.

```
contract example {
    modifier check_price(int64 price) {
        if (price >= 50) {
            _;
        }
    }

    function foo(int64 price) check_price(price) public {
        // ...
    }
}
```

Multiple modifiers can be applied to single function. The modifiers are executed in the order of the modifiers specified on the function declaration. Execution will continue to the next modifier when the `_;` is reached. In this example, the *only_owner* modifier is run first, and if that reaches `_;`, then *check_price* is executed. The body of function *foo()* is only reached once *check_price()* reaches `_;`.

```
contract example {
    address owner;

    // a modifier with no arguments does not need "()" in its declaration
    modifier only_owner {
        require(msg.sender == owner);
        _;
    }

    modifier check_price(int64 price) {
        if (price >= 50) {
            _;
        }
    }

    function foo(int64 price) only_owner check_price(price) public {
        // ...
    }
}
```

Modifiers can be inherited or declared `virtual` in a base contract and then overriden, exactly like functions can be.

```
contract base {
    address owner;

    modifier only_owner {
        require(msg.sender == owner);
        _;
    }
```

(continues on next page)

(continued from previous page)

```
    modifier check_price(int64 price) virtual {
        if (price >= 10) {
            _;
        }
    }
}

contract example is base {
    modifier check_price(int64 price) override {
        if (price >= 50) {
            _;
        }
    }

    function foo(int64 price) only_owner check_price(price) public {
        // ...
    }
}
```

4.5.6 Calling an external function using `call()`

If you call a function on a contract, then the function selector and any arguments are ABI encoded for you, and any return values are decoded. Sometimes it is useful to call a function without abi encoding the arguments.

You can call a contract directly by using the `call()` method on the address type. This takes a single argument, which should be the ABI encoded arguments. The return values are a `boolean` which indicates success if true, and the ABI encoded return value in `bytes`.

```
contract a {
    function test() public {
        b v = new b();

        // the following four lines are equivalent to "uint32 res = v.foo(3,5);"

        // Note that the signature is only hashed and not parsed. So, ensure that the
        // arguments are of the correct type.
        bytes data = abi.encodeWithSignature("foo(uint32,uint32)", uint32(3),␣
↪uint32(5));

        (bool success, bytes rawresult) = address(v).call(data);

        assert(success == true);

        uint32 res = abi.decode(rawresult, (uint32));

        assert(res == 8);
    }
}

contract b {
    function foo(uint32 a, uint32 b) public returns (uint32) {
        return a + b;
    }
}
```

Any value or gas limit can be specified for the external call. Note that no check is done to see if the called function is

`payable`, since the compiler does not know what function you are calling.

```
function test(address foo, bytes rawcalldata) public {
    (bool success, bytes rawresult) = foo.call{value: 102, gas: 1000}(rawcalldata);
}
```

Note: ewasm also supports `staticcall()` and `delegatecall()` on the address type. These call types are not supported on Parity Substrate.

4.5.7 fallback() and receive() function

When a function is called externally, either via an transaction or when one contract call a function on another contract, the correct function is dispatched based on the function selector in the raw encoded ABI call data. If there is no match, the call reverts, unless there is a `fallback()` or `receive()` function defined.

If the call comes with value, then `receive()` is executed, otherwise `fallback()` is executed. This made clear in the declarations; `receive()` must be declared `payable`, and `fallback()` must not be declared `payable`. If a call is made with value and no `receive()` function is defined, then the call reverts, likewise if call is made without value and no `fallback()` is defined, then the call also reverts.

Both functions must be declared `external`.

```
contract test {
    int32 bar;

    function foo(uint32 x) public {
        bar = x;
    }

    fallback() external {
        // execute if function selector does not match "foo(uint32)" and no value sent
    }

    receive() payable external {
        // execute if function selector does not match "foo(uint32)" and value sent
    }
}
```

4.6 Constants

Constants can be declared at the global level or at the contract level, just like contract storage variables. They do not use any contract storage and cannot be modified. The variable must have an initializer, which must be a constant expression. It is not allowed to call functions or read variables in the initializer:

```
string constant greeting = "Hello, World!";

contract ethereum {
    uint constant byzantium_block = 4_370_000;
}
```

4.7 Contract Storage

Any variables declared at the contract level (so not declared in a function or constructor), will automatically become contract storage. Contract storage is maintained on chain, so they retain their values between calls. These are declared so:

```
contract hitcount {
    uint counter = 1;

    function hit() public {
        counters++;
    }

    function count() public view returns (uint) {
        return counter;
    }
}
```

The `counter` is maintained for each deployed `hitcount` contract. When the contract is deployed, the contract storage is set to 1. Contract storage variable do not need an initializer; when it is not present, it is initialized to 0, or `false` if it is a `bool`.

4.7.1 How to clear Contract Storage

Any contract storage variable can have its underlying contract storage cleared with the `delete` operator. This can be done on any type; a simple integer, an array element, or the entire array itself. Contract storage has to be cleared slot (i.e. primitive) at a time, so if there are many primitives, this can be costly.

```
contract s {
    struct user {
        address f1;
        int[] list;
    }
    user[1000] users;

    function clear() public {
        // delete has to iterate over 1000 users, and for each of those clear the
        // f1 field, read the length of the list, and iterate over each of those
        delete users;
    }
}
```

4.8 Events

In Solidity, contracts can emit events that signal that changes have occurred. For example, a Solidity contract could emit a *Deposit* event, or *BetPlaced* in a poker game. These events are stored in the blockchain transaction log, so they become part of the permanent record. From Solidity's perspective, you can emit events but you cannot access events on the chain.

Once those events are added to the chain, an off-chain application can listen for events. For example, the Web3.js interface has a *subscribe()* function. Another is example is Hyperledger Burrow which has a vent command which listens to events and inserts them into a Postgres database.

An event has two parts. First, there is a limited set of topics. Usually there are no more than 3 topics, and each of those has a fixed length of 32 bytes. They are there so that an application listening for events can easily filter for particular types of events, without needing to do any decoding. There is also a data section of variable length bytes, which is ABI encoded. To decode this part, the ABI for the event must be known.

From Solidity's perspective, an event has a name, and zero or more fields. The fields can either be `indexed` or not. `indexed` fields are stored as topics, so there can only be a limited number of `indexed` fields. The other fields are stored in the data section of the event. The event name does not need to be unique; just like functions, they can be overloaded as long as the fields are of different types, or the event has a different number of arguments. In Parity Substrate, the topic fields are always the hash of the value of the field. Ethereum only hashes fields which do not fit in the 32 bytes. Since a cryptographic hash is used, it is only possible to compare the topic against a known value.

An event can be declared in a contract, or outside.

```
event CounterpartySigned (
    address indexed party,
    address counter_party,
    uint contract_no
);

contract Signer {
    funtion sign(address counter_party, uint contract_no) internal {
        emit CounterpartySigned(address(this), counter_party, contract_no);
    }
}
```

Like function calls, the emit statement can have the fields specified by position, or by field name. Using field names rather than position may be useful in case the event name is overloaded, since the field names make it clearer which exact event is being emitted.

```
event UserModified(
    address user,
    string name
) anonymous;

event UserModified(
    address user,
    uint64 groupid
);

contract user {
    function set_name(string name) public {
        emit UserModified({ user: msg.sender, name: name });
    }

    function set_groupid(uint64 id) public {
        emit UserModified({ user: msg.sender, groupid: id });
    }
}
```

In the transaction log, the first topic of an event is the keccak256 hash of the signature of the event. The signature is the event name, followed by the fields types in a comma separated list in parentheses. So the first topic for the second UserModified event would be the keccak256 hash of `UserModified(address,uint64)`. You can leave this topic out by declaring the event `anonymous`. This makes the event slightly smaller (32 bytes less) and makes it possible to have 4 `indexed` fields rather than 3.

4.9 Constructors and contract instantiation

When a contract is deployed, the contract storage is initialized to the initializer values provided, and any constructor is called. A constructor is not required for a contract. A constructor is defined like so:

```
contract mycontract {
    uint foo;

    constructor(uint foo_value) {
        foo = foo_value;
    }
}
```

A constructor does not have a name and may have any number of arguments. If a constructor has arguments, then when the contract is deployed then those arguments must be supplied.

If a contract is expected to hold receive value on instantiation, the constructor should be declared `payable`.

Note: Parity Substrate allows multiple constructors to be defined, which is not true for ewasm. So, when building for Substrate, multiple constructors can be defined as long as their argument list is different (i.e. overloaded).

When the contract is deployed in the Polkadot UI, the user can select the constructor to be used.

4.9.1 Instantiation using new

Contracts can be created using the `new` keyword. The contract that is being created might have constructor arguments, which need to be provided.

```
contact hatchling {
    string name;

    constructor(string id) {
        require(id != "", "name must be provided");
        name = id;
    }
}

contract adult {
    function test() public {
        hatchling h = new hatchling("luna");
    }
}
```

The constructor might fail for various reasons, for example `require()` might fail here. This can be handled using the *Try Catch Statement* statement, else errors cause the transaction to fail.

4.9.2 Sending value to the new contract

It is possible to send value to the new contract. This can be done with the `{value: 500}` syntax, like so:

```
contact hatchling {
    string name;
```

(continues on next page)

(continued from previous page)

```
    constructor(string id) payable {
        require(id != "", "name must be provided");
        name = id;
    }
}

contract adult {
    function test() public {
        hatchling h = new hatchling{value: 500}("luna");
    }
}
```

The constructor should be declared `payable` for this to work.

Note: If no value is specified, then on Parity Substrate the minimum balance (also know as the existential deposit) is sent.

4.9.3 Setting the salt and gas for the new contract

Note: ewasm does not yet provide a method for setting the salt or gas for the new contract, so these values are ignored.

When a new contract is created, the address for the new contract is a hash of the input (the constructor arguments) to the new contract. So, a contract cannot be created twice with the same input. This is why the salt is concatenated to the input. The salt is either a random value or it can be explicitly set using the `{salt: 2}` syntax. A constant will remove the need for the runtime random generation, however creating a contract twice with the same salt and arguments will fail. The salt is of type `uint256`.

If gas is specified, this limits the amount gas the constructor for the new contract can use. gas is a `uint64`.

```
contact hatchling {
    string name;

    constructor(string id) payable {
        require(id != "", "name must be provided");
        name = id;
    }
}

contract adult {
    function test() public {
        hatchling h = new hatchling{salt: 0, gas: 10000}("luna");
    }
}
```

4.10 Base contracts, abstract contracts and interfaces

Solidity contracts support object-oriented programming. The style Solidity is somewhat similar to C++, but there are many differences. In Solidity we are dealing with contracts, not classes.

4.10.1 Specifying base contracts

To inherit from another contract, you have to specify it as a base contract. Multiple contracts can be specified here.

```
contact a is b, c {
    constructor() {}
}

contact b {
    int foo;
    function func2() public {}
    constructor() {}
}

contact c {
    int bar;
    constructor() {}
    function func1() public {}
}
```

In this case, contract `a` inherits from both `b` and `c`. Both `func1()` and `func2()` are visible in contract `a`, and will be part of its public interface if they are declared `public` or `external`. In addition, the contract storage variables `foo` and `bar` are also availabe in `a`.

Inheriting contracts is recursive; this means that if you inherit a contract, you also inherit everything that that contract inherits. In this example, contract `a` inherits `b` directly, and inherits `c` through `b`. This means that contract `b` also has a variable `bar`.

```
contact a is b {
    constructor() {}
}

contact b is c {
    int foo;
    function func2() public {}
    constructor() {}
}

contact c {
    int bar;
    constructor() {}
    function func1() public {}
}
```

4.10.2 Virtual Functions

When inheriting from a base contract, it is possible to override a function with a newer function with the same name. For this to be possible, the base contract must have specified the function as `virtual`. The inheriting contract must then specify the same function with the same name, arguments and return values, and add the `override` keyword.

```
contact a is b {
    function func(int a) override public returns (int) {
        return a + 11;
    }
}
```

(continues on next page)

(continued from previous page)

```
contact b {
    function func(int a) virtual public returns (int) {
        return a + 10;
    }
}
```

If the function is present in more than one base contract, the `override` attribute must list all the base contracts it is overriding.

```
contact a is b,c {
    function func(int a) override(b,c) public returns (int) {
        return a + 11;
    }
}

contact b {
    function func(int a) virtual public returns (int) {
        return a + 10;
    }
}

contact c {
    function func(int a) virtual public returns (int) {
        return a + 5;
    }
}
```

4.10.3 Calling function in base contract

When a virtual function is called, the dispatch is *virtual*. If the function being called is overriden in another contract, then the overriding function is called. For example:

```
contract a is b {
    function baz() public returns (uint64) {
        return foo();
    }

    function foo() internal override returns (uint64) {
        return 2;
    }
}

contract a {
    function foo() internal virtual returns (uint64) {
        return 1;
    }

    function bar() internal returns (uint64) {
        // since foo() is virtual, is a virtual dispatch call
        // when foo is called and a is a base contract of b, then foo in contract b␣
↪will
        // be called; foo will return 2.
        return foo();
    }
```

(continues on next page)

(continued from previous page)

```
    function bar2() internal returns (uint64) {
        // this explicitly says "call foo of base contract a", and dispatch is not␣
↪virtual
        return a.foo();
    }
}
```

Rather than specifying the base contract, use `super` as the contract to call the base contract function.

```
contract a is b {
    function baz() public returns (uint64) {
        // this will return 1
        return super.foo();
    }

    function foo() internal override returns (uint64) {
        return 2;
    }
}

contract b {
    function foo() internal virtual returns (uint64) {
        return 1;
    }
}
```

If there are multiple base contracts which the define the same function, the function of the first base contract is called.

```
 contract a is b1, b2 {
     function baz() public returns (uint64) {
         // this will return 100
         return super.foo();
     }

     function foo() internal override(b2, b2) returns (uint64) {
         return 2;
     }
 }

 contract b1 {
     function foo() internal virtual returns (uint64) {
         return 100;
     }
}

 contract b2 {
     function foo() internal virtual returns (uint64) {
         return 200;
     }
 }
```

4.10.4 Specifying constructor arguments

If a contract inherits another contract, then when it is instantiated or deployed, then the constructor for its inherited contracts is called. The constructor arguments can be specified on the base contract itself.

```
contact a is b(1) {
    constructor() {}
}

contact b is c(2) {
    int foo;
    function func2(int i) public {}
    constructor() {}
}

contact c {
    int bar;
    constructor(int32 j) {}
    function func1() public {}
}
```

When `a` is deployed, the constructor for `c` is executed first, then `b`, and lastly `a`. When the constructor arguments are specified on the base contract, the values must be constant. It is possible to specify the base arguments on the constructor for inheriting contract. Now we have access to the constructor arguments, which means we can have runtime-defined arguments to the inheriting constructors.

```
contact a is b {
    constructor(int i) b(i+2) {}
}

contact b is c {
    int foo;
    function func2() public {}
    constructor(int j) c(j+3) {}
}

contact c {
    int bar;
    constructor(int32 k) {}
    function func1() public {}
}
```

The execution is not entirely intuitive in this case. When contract `a` is deployed with an int argument of 10, then first the constructor argument or contract `b` is calculated: 10+2, and that value is used as an argument to constructor `b`. constructor `b` calculates the arguments for constructor `c` to be: 12+3. Now, with all the arguments for all the constructors established, constructor `c` is executed with argument 15, then constructor `b` with argument 12, and lastly constructor `a` with the original argument 10.

4.10.5 Abstract Contracts

An `abstract contract` is one that cannot be instantiated, but it can be used as a base for another contract, which can be instantiated. A contract can be abstract because the functions it defines do not have a body, for example:

```
abstract contact a {
    function func2() virtual public;
}
```

This contract cannot be instantiated, since there is no body or implementation for `func2`. Another contract can define this contract as a base contract and override `func2` with a body.

Another reason why a contract must be abstract is missing constructor arguments. In this case, if we were to instantiate

contract `a` we would not know what the constructor arguments to its base `b` would have to be. Note that contract `c` does inherit from `a` and can specify the arguments for `b` on its constructor, even though `c` does not directly inherit `b` (but does indirectly).

```
abstract contact a is b {
    constructor() {}
}

contact b {
    constructor(int j) {}
}

contract c is a {
    constructor(int k) b(k*2) {}
}
```

4.10.6 Interfaces

An interface is a contract sugar type with restrictions. This type cannot be instantiated; it can only define the functions prototypes for a contract. This is useful as a generic interface.

```
interface operator {
    function op1(int32 a, int32 b) external returns (int32);
    function op2(int32 a, int32 b) external returns (int32);
}

contract ferqu {
    operator op;

    constructor(bool do_adds) {
        if (do_adds) {
            op = new m1();
        } else {
            op = new m2();
        }
    }

    function x(int32 b) public returns (int32) {
        return op.op1(102, b);
    }
}

contract m1 is operator {
    function op1(int32 a, int32 b) public override returns (int32) {
        return a + b;
    }

    function op2(int32 a, int32 b) public override returns (int32) {
        return a - b;
    }
}

contract m2 is operator {
    function op1(int32 a, int32 b) public override returns (int32) {
        return a * b;
    }
```

(continues on next page)

(continued from previous page)

```
    function op2(int32 a, int32 b) public override returns (int32) {
        return a / b;
    }
}
```

- Interfaces can only have other interfaces as a base contract
- All functions must the `external` visibilty
- No constructor can be declared
- No contract storage variables can exist (however constants are allowed)
- No function can have a body or implementation

4.10.7 Libraries

Libraries are a special type of contract which can be reused in multiple contracts. Functions declared in a library can be called with the `library.function()` syntax. When the library has been imported or declared, any contract can use its functions simply by using its name.

```
contract test {
    function foo(uint64 x) public pure returns (uint64) {
        return ints.max(x, 65536);
    }
}

library ints {
    function max(uint64 a, uint64 b) public pure returns (uint64) {
        return a > b ? a : b;
    }
}
```

When writing libraries there are restrictions compared to contracts:

- A library cannot have constructors, fallback or receive function
- A library cannot have base contracts
- A library cannot be a base contract
- A library cannot have virtual or override functions
- A library cannot have payable functions

Note: When using the Ethereum Foundation Solidity compiler, library are a special contract type and libraries are called using *delegatecall*. Parity Substrate has no `delegatecall` functionality so Solang statically links the library calls into your contract code. This does make for larger contract code, however this reduces the call overhead and make it possible to do compiler optimizations across library and contract code.

4.10.8 Library Using For

Libraries can be used as method calls on variables. The type of the variable needs to be bound to the library, and the type of the first parameter of the function of the library must match the type of a variable.

```
contract test {
    using lib for int32[100];

    int32[100] bar;

    function foo() public returns (int64) {
        bar.set(10, 571);
    }
}

library lib {
    function set(int32[100] storage a, uint index, int32 val) internal {
        a[index] = val;
    }
}
```

The syntax `using` *library* `for` *Type* `;` is the syntax that binds the library to the type. This must be specified on the contract. This binds library `lib` to any variable with type `int32[100]`. As a result of this, any method call on a variable of type `int32[100]` will be matched to library `lib`.

For the call to match, the first argument of the function must match the variable; note that here, *bar* is of type `storage`, since all contract variables are implicitly `storage`.

There is an alternative syntax `using` *library* `for *;` which binds the library functions to any variable that will match according to these rules.

4.11 Sending and receiving value

Value in Solidity is represented by `uint128`.

Note: Parity Substrate can be compiled with a different type for `T::Balance`. If you need support for a different type, please raise an issue.

4.11.1 Checking your balance

The balance of a contract can be checked with *address* `.balance`, so your own balance is `address(this).balance`.

Note: Parity Substrate cannot check the balance for contracts other than the current one. If you need to check the balance of another contract, then add a balance function to that contract like the one below, and call that function instead.

```
function balance() public returns (uint128) {
    return address(this).balance;
}
```

4.11.2 Creating contracts with an initial value

You can specify the value you want to be deposited in the new contract by specifying `{value: 100 ether}` before the constructor arguments. This is explained in *sending value to the new contract*.

4.11.3 Sending value with an external call

You can specify the value you want to be sent along with the function call by specifying `{value: 100 ether}` before the function arguments. This is explained in *passing value and gas with external calls*.

4.11.4 Sending value using `send()` and `transfer()`

The `send()` and `transfer()` functions are available as method on a `address payable` variable. The single arguments is the amount of value you would like to send. The difference between the two functions is what happens in the failure case: `transfer()` will revert the current call, `send()` returns a `bool` which will be `false`.

In order for the receiving contract to receive the value, it needs a `receive()` function, see *fallback() and receive() function*.

Here is an example:

```
contract A {
    B other;

    constructor() {
        other = new B();

        bool complete = payable(other).transfer(100);

        if (!complete) {
            // oops
        }

        // if the following fails, our transaction will fail
        other.send(100);
    }
}

contract B {
    receive() payable external {
        // ..
    }
}
```

Note: On Subtrate, this uses the `seal_transfer()` mechanism rather than `seal_call()`, since this does not come with gas overhead. This means the `receive()` function is not required in the receiving contract, and it will not be called if it is present. If you want the `receive()` function to be called, use `address.call{value: 100}("")` instead.

4.12 Builtin Functions and Variables

The Solidity language has a number of built-in variables and functions which give access to the chain environment or pre-defined functions. Some of these functions will be different on different chains.

4.12.1 Block and transaction

The functions and variables give access to block properties like block number and transaction properties like gas used, and value sent.

gasleft() returns (uint64)

Returns the amount of gas remaining the current transaction.

blockhash(uint64 block) returns (bytes32)

Returns the blockhash for a particular block. This not possible for the current block, or any block except for the most recent 256. Do not use this a source of randomness unless you know what you are doing.

Note: This function is not available on Parity Substrate. When using Parity Substrate, use `random()` as a source of random data.

---

random(bytes subject) returns (bytes32)

Returns random bytes based on the subject. The same subject for the same transaction will return the same random bytes, so the result is deterministic. The chain has a `max_subject_len`, and if *subject* exceeds that, the transaction will be aborted.

Note: This function is only available on Parity Substrate.

`msg` properties

uint128 `msg.value` The amount of value sent with a transaction, or 0 if no value was sent.

bytes `msg.data` The raw ABI encoded arguments passed to the current call.

bytes4 `msg.sig` Function selector from the ABI encoded calldata, e.g. the first four bytes. This might be 0 if no function selector was present. In Ethereum, constructor calls do not have function selectors but in Parity Substrate they do.

address `msg.sender` The sender of the current call. This is either the address of the contract that called the current contract, or the address that started the transaction if it called the current contract directly.

`tx` properties

uint128 `tx.gasprice` The price of one unit of gas. This field cannot be used on Parity Substrate, the explanation is in the warning box below.

uint128 `tx.gasprice(uint64 gas)` The total price of *gas* units of gas.

Warning: On Parity Substrate, the cost of one gas unit may not be an exact whole round value. In fact, if the gas price is less than 1 it may round down to 0, giving the incorrect appearance gas is free. Therefore, avoid the `tx.gasprice` member in favour of the function `tx.gasprice(uint64 gas)`.

To avoid rounding errors, pass the total amount of gas into `tx.gasprice(uint64 gas)` rather than doing arithmetic on the result. As an example, **replace** this bad example:

```
// BAD example
uint128 cost = num_items * tx.gasprice(gas_per_item);
```

with:

```
uint128 cost = tx.gasprice(num_items * gas_per_item);
```

Note this function is not available on the Ethereum Foundation Solidity compiler.

address `tx.origin` The address that started this transaction. Not available on Parity Substrate

`block` properties

Some block properties are always available:

uint64 `block.number` The current block number.

uint64 `block.timestamp` The time in unix epoch, i.e. seconds since the beginning of 1970.

Do not use either of these two fields as a source of randomness unless you know what you are doing.

The other block properties depend on which chain is being used.

Parity Substrate

uint128 `block.tombstone_deposit` The amount needed for a tombstone. Without it, contracts will disappear completely if the balance runs out.

uint128 `block.minimum_deposit` The minimum amonut needed to create a contract. This does not include storage rent.

Ethereum

uint64 `block.gaslimit` The current block gas limit.

address payable `block.coinbase` The current block miner's address.

uint256 `block.difficulty` The current block's difficulty.

4.12.2 Error handling

assert(bool)

Assert takes a boolean argument. If that evaluates to false, execution is aborted.

```
contract c {
    constructor(int x) {
        assert(x > 0);
    }
}
```

revert() or revert(string)

revert aborts execution of the current contract, and returns to the caller. revert() can be called with no arguments, or a single *string* argument, which is called the *ReasonCode*. This function can be called at any point, either in a constructor or a function.

If the caller is another contract, it can use the *ReasonCode* in a *Try Catch Statement* statement.

```
contract x {
    constructor(address foobar) {
        if (a == address(0)) {
            revert("foobar must a valid address");
        }
    }
}
```

require(bool) or require(bool, string)

This function is used to check that a condition holds true, or abort execution otherwise. So, if the first *bool* argument is *true*, this function does nothing, however if the *bool* arguments is *false*, then execution is aborted. There is an optional second *string* argument which is called the *ReasonCode*, which can be used by the caller to identify what the problem is.

```
contract x {
    constructor(address foobar) {
        require(foobar != address(0), "foobar must a valid address");
    }
}
```

4.12.3 ABI encoding and decoding

The ABI encoding depends on the target being compiled for. Substrate uses the SCALE Codec and ewasm uses Ethereum ABI encoding.

abi.decode(bytes, (*type-list*))

This function decodes the first argument and returns the decoded fields. *type-list* is a comma-separated list of types. If multiple values are decoded, then a destructure statement must be used.

```
uint64 foo = abi.decode(bar, (uint64));
```

```
(uint64 foo1, bool foo2) = abi.decode(bar, (uint64, bool));
```

If the arguments cannot be decoded, contract execution will abort. This can happen if the encoded length is too short, for example.

abi.encode(...)

ABI encodes the arguments to bytes. Any number of arguments can be provided.

```
uint16 x = 241;
bytes foo = abi.encode(x);
```

On Substrate, foo will be `hex"f100"`. On Ethereum this will be `hex"00000000000000000000000000000000000000000000000000000000000000f1"`.

abi.encodeWithSelector(bytes4 selector, ...)

ABI encodes the arguments with the function selector first. After the selector, any number of arguments can be provided.

```
bytes foo = abi.encodeWithSelector(hex"01020304", uint16(0xff00), "ABCD");
```

On Substrate, foo will be `hex"0403020100ff"`. On Ethereum this will be `hex"0102030400000000000000000000000000000000000000000000000000000000000000ff00"`.

abi.encodeWithSignature(string signature, ...)

ABI encodes the arguments with the `bytes4` hash of the signature. After the signature, any number of arguments can be provided. This is equivalent to `abi.encodeWithSignature(bytes4(keccak256(signature)), ...)`.

```
bytes foo = abi.encodeWithSignature("test2(uint64)", uint64(257));
```

On Substrate, foo will be `hex"296dacf0_0101_0000__0000_0000"`. On Ethereum this will be `hex"296dacf0_0000000000000000000000000000000000000000000000000000000000000101"`.

abi.encodePacked(...)

ABI encodes the arguments to bytes. Any number of arguments can be provided. The packed encoding only encodes the raw data, not the lengths of strings and arrays. For example, when encoding `string` only the string bytes will be encoded, not the length. It is not possible to decode packed encoding.

```
bytes foo = abi.encode(uint16(0xff00), "ABCD");
```

On Substrate, foo will be `hex"00ff41424344"`. On Ethereum this will be `hex"ff0041424344"`.

4.12.4 Cryptography

keccak256(bytes)

This returns the `bytes32` keccak256 hash of the bytes.

ripemd160(bytes)

This returns the `bytes20` ripemd160 hash of the bytes.

sha256(bytes)

This returns the `bytes32` sha256 hash of the bytes.

blake2_128(bytes)

This returns the `bytes16` blake2_128 hash of the bytes.

Note: This function is only available on Parity Substrate.

blake2_256(bytes)

This returns the `bytes32` blake2_256 hash of the bytes.

Note: This function is only available on Parity Substrate.

4.12.5 Mathematical

addmod(uint x, uint y, uint, k) returns (uint)

Add x to y, and then divides by k. x + y will not overflow.

mulmod(uint x, uint y, uint, k) returns (uint)

Multiply x with y, and then divides by k. x * y will not overflow.

4.12.6 Miscellaneous

print(string)

print() takes a string argument.

```
contract c {
    constructor() {
        print("Hello, world!");
    }
}
```

Note: print() is not available with the Ethereum Foundation Solidity compiler.

When using Substrate, this function is only available on development chains. If you use this function on a production chain, the contract will fail to load.

When using ewasm, the function is only available on hera when compiled with debugging.

selfdestruct(address payable recipient)

The `selfdestruct()` function causes the current contract to be deleted, and any remaining balance to be sent to *recipient*. This functions does not return, as the contract no longer exists.

String formatting using `"{}".format()`

Sometimes it is useful to convert an integer to a string, e.g. for debugging purposes. There is a format builtin function for this, which is a method on string literals. Each `{}` in the string will be replaced with the value of an argument to format().

```
function foo(int arg1, bool arg2) public {
    print("foo entry arg1:{} arg2:{}".format(arg1, arg2));
}
```

Assuming *arg1* is 5355 and *arg2* is true, the output to the log will be `foo entry arg1:5355 arg2:true`.

The types accepted by format are `bool`, `uint`, `int` (any size, e.g. `int128` or `uint64`), `address`, `bytes` (fixed and dynamic), and `string`. Enums are also supported, but will print the ordinal value of the enum. The `uint` and `int` types can have a format specifier. This allows you to convert to hexadecimal `{:x}` or binary `{:b}`, rather than decimals. No other types have a format specifier. To include a literal `{` or `}`, replace it with `{{` or `}}`.

```
function foo(int arg1, uint arg2) public {
    // print arg1 in hex, and arg2 in binary
    print("foo entry {{arg1:{:x},arg2:{:b}}}".format(arg1, arg2));
}
```

Assuming *arg1* is 512 and *arg2* is 196, the output to the log will be `foo entry {arg1:0x200, arg2:0b11000100}`.

Warning: Each time you call the `format()` some specialized code is generated, to format the string at runtime. This requires loops and so on to do the conversion.

When formatting integers in to decimals, types larger than 64 bits require expensive division. Be mindful this will increase the gas cost. Larger values will incur a higher gas cost. Alternatively, use a hexadecimal `{:x}` format specifier to reduce the cost.

4.13 Tags

Any contract, interface, library, event definition, struct definition, function, or contract variable may have tags associated with them. These are used for generating documentation for the contracts, when Solang is run with the `--doc` command line option. This option generates some html which lists all the types, contracts, functions, and state variables along with their tags.

The tags use a special comment format. They can either be specified in block comments or single line comments.

```
/**
 * @title Hello, World!
 * @notice Just an example.
 * @author Sean Young <sean@mess.org>
 */
contract c {
    /// @param name The name which will be greeted
    function say_hello(string name) {
        print("Hello, " + name + "!");
    }
}
```

The tags which are allowed:

`@title` Headline for this unit

`@notice` General body for explaining what this unit does

`@dev` Any development notes

`@author` Field for the author of this code

`@param` ***name*** Document a function parameter, field of struct or event. Requires a name of the field or parameter

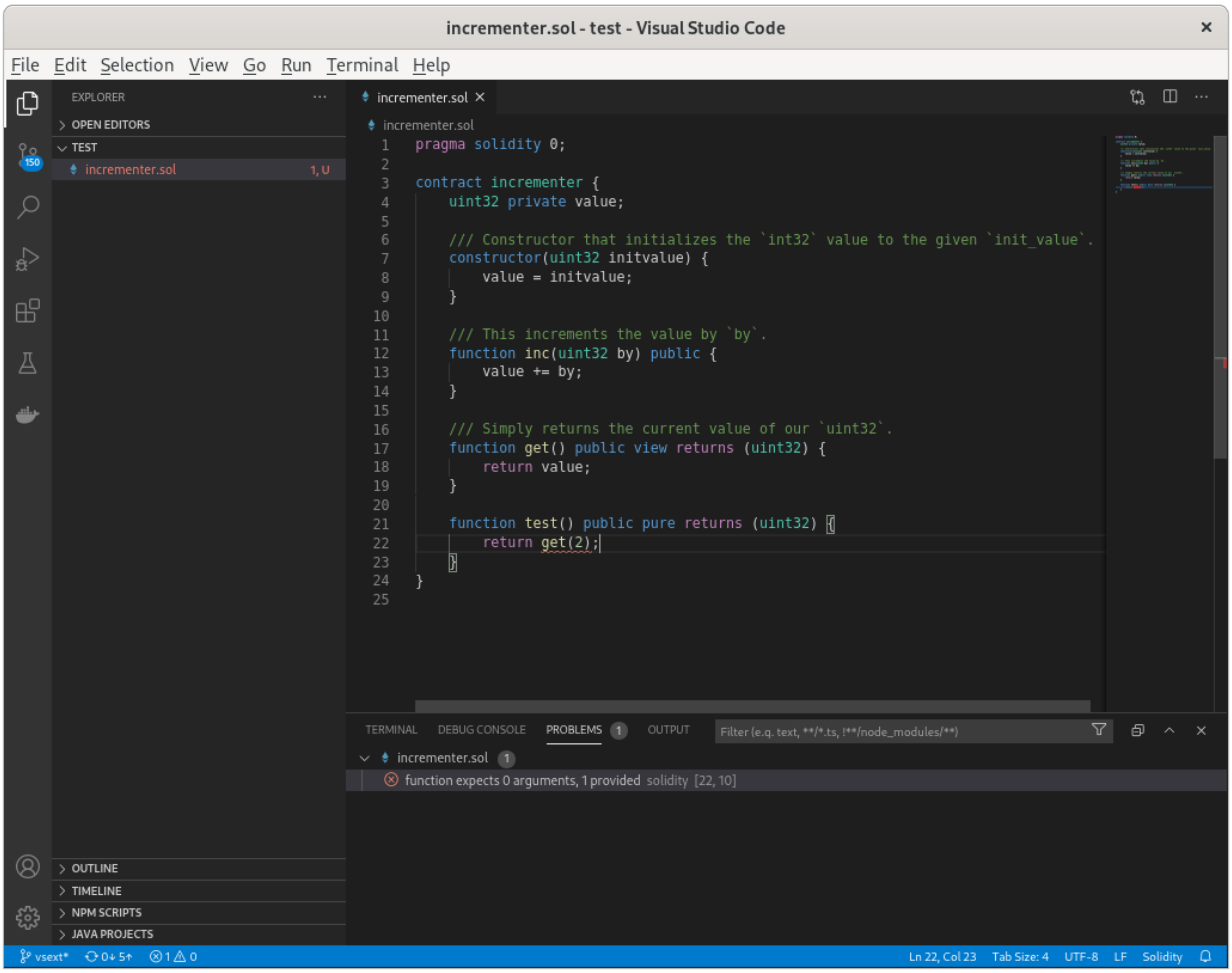
`@return` ***name*** Document a function return value. Requires a name of the field or parameter if the function returns more than one value.

CHAPTER 5

Visual Studio Code Extension

Solang has language server built into the executable, which can be used by the Visual Studio Code extension. This extension provides the following:

1. Syntax highlighting
2. Compiler warnings and errors are displayed in the problems tab and marked with squiqqly lines, this is also known as *diagnostics*.
3. Hovering over variables, types, functions etc and more will give information, For example this will give the struct fields when hovering over a variable which is a reference to a struct.

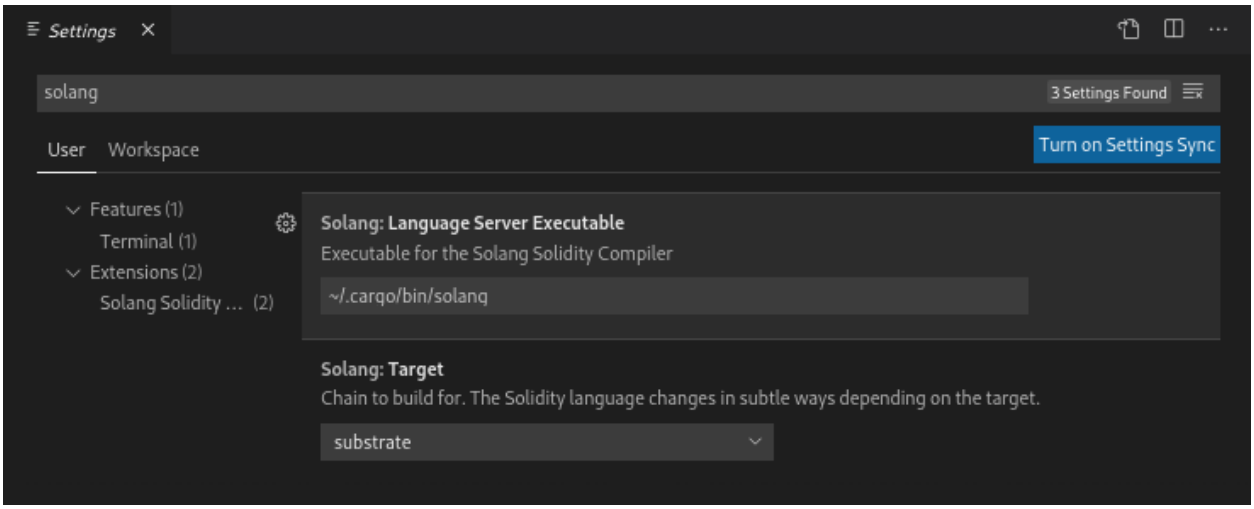


Both the Visual Studio Code extension code and the language server were developed under a Hyperledger Mentorship programme.

5.1 Using the extension

The extension can be found on the Visual Studio Marketplace.

First, install the extension and the Solang compiler binary. The extension needs to know where to find the Solang binary to start the language server, and also it needs to know what target you wish to compile your solidity code for.



5.2 Development

The code is spread over two parts. The first part the vscode extension client code, written in TypeScript. This part deals with syntax highlighting, and calling out to the Solang language server when needed. The client needs npm and node installed. The client implementation is present in src/client. The extension client code is in src/client/extension.ts.

Secondly, there is the language server which is written in Rust. The Solang binary has an option `--language-server`, which start the built-in language server.

Once you have node and npm installed, you can build the extension like so:

```
git clone https://github.com/hyperledger-labs/solang
cd solang/vscode
npm install
npm install -g vsce
vsce package
```

You should now have an extension file called solang-0.0.1.vsix which can be installed using *code –install-extension solang-0.0.1.vsix*.

Alternatively, the extension be run from vscode itself.

1. Inside a vscode instance, Ctrl+Shift+B to build the project
2. On the task bar at the bottom of the IDE select Launch Client
3. Open a Solidity file (.sol) to test the extension.

To run the tests:

1. Inside a vscode instance, Ctrl+Shift+B to build the project
2. On the task bar at the bottom of the IDE select Extensions tests
3. The result should be displayed in the debug console of the host IDE instance.

CHAPTER 6

Solang Solidity Examples

Here are two examples of Solidity contracts.

6.1 Flipper

This is the ink! flipper example written in Solidity:

```
contract flipper {
        bool private value;

        /// Constructor that initializes the `bool` value to the given `init_value`.
        constructor(bool initvalue) {
                value = initvalue;
        }

        /// A message that can be called on instantiated contracts.
        /// This one flips the value of the stored `bool` from `true`
        /// to `false` and vice versa.
        function flip() public {
                value = !value;
        }

        /// Simply returns the current value of our `bool`.
        function get() public view returns (bool) {
                return value;
        }
}
```

6.2 Full Example

This example exists to show the language features that Solang supports.

```
// full_example.sol

/*
 This is an example contract to show all the features that the
 Solang Solidity Compiler supports.
*/

contract full_example {
        // Process state
        enum State {
                Running,
                Sleeping,
                Waiting,
                Stopped,
                Zombie,
                StateCount
        }

        // Variables in contract storage
        State state;
        int32 pid;
        uint32 reaped = 3;

        // Constants
        State constant bad_state = State.Zombie;
        int32 constant first_pid = 1;

        // Our constructors
        constructor(int32 _pid) {
                // Set contract storage
                pid = _pid;
        }

        // Reading but not writing contract storage means function
        // can be declared view
        function is_zombie_reaper() public view returns (bool) {
                /* must be pid 1 and not zombie ourselves */
                return (pid == first_pid && state != State.Zombie);
        }

        // Returning a constant does not access storage at all, so
        // function can be declared pure
        function systemd_pid() public pure returns (uint32) {
                // Note that cast is required to change sign from
                // int32 to uint32
                return uint32(first_pid);
        }

        /// Convert celcius to fahrenheit
        function celcius2fahrenheit(int32 celcius) pure public returns (int32) {
                int32 fahrenheit = celcius * 9 / 5 + 32;

                return fahrenheit;
        }

        /// Convert fahrenheit to celcius
        function fahrenheit2celcius(int32 fahrenheit) pure public returns (int32) {
```

(continues on next page)

(continued from previous page)

```
                return (fahrenheit - 32) * 5 / 9;
        }

        /// is this number a power-of-two
        function is_power_of_2(uint n) pure public returns (bool) {
                return n != 0 && (n & (n - 1)) == 0;
        }

        /// calculate the population count (number of set bits) using Brian Kerningham
↪'s way
        function population_count(uint n) pure public returns (uint count) {
                for (count = 0; n != 0; count++) {
                        n &= (n - 1);
                }
        }

        /// calculate the power of base to exp
        function power(uint base, uint exp) pure public returns (uint) {
                return base ** exp;
        }

        /// returns true if the address is 0
        function is_address_zero(address a) pure public returns (bool) {
                return a == address(0);
        }

        /// reverse the bytes in an array of 8 (endian swap)
        function byte8reverse(bytes8 input) public pure returns (bytes8 out) {
                out = ((input << 56) & hex"ff00_0000_0000_0000") |
                          ((input << 40) & hex"00ff_0000_0000_0000") |
                          ((input << 24) & hex"0000_ff00_0000_0000") |
                          ((input <<  8) & hex"0000_00ff_0000_0000") |
                          ((input >>  8) & hex"0000_0000_ff00_0000") |
                          ((input >> 24) & hex"0000_0000_00ff_0000") |
                          ((input >> 40) & hex"0000_0000_0000_ff00") |
                          ((input >> 56) & hex"0000_0000_0000_00ff");
        }

        /// This mocks a pid state
        function get_pid_state(int64 _pid) pure private returns (State) {
                int64 n = 8;
                for (int16 i = 1; i < 10; ++i) {
                        if ((i % 3) == 0) {
                                n *= _pid / int64(i);
                        } else {
                                n /= 3;
                        }
                }

                return State(n % int64(State.StateCount));
        }

        /// Overloaded function with different return value!
        function get_pid_state() view private returns (uint32) {
                return reaped;
        }
```

(continues on next page)

(continued from previous page)

```
        function reap_processes() public {
                int32 n = 0;

                while (n < 100) {
                        if (get_pid_state(n) == State.Zombie) {
                                // reap!
                                reaped += 1;
                        }
                        n++;
                }
        }

        function run_queue() public pure returns (uint16) {
                uint16 count = 0;
                // no initializer means its 0.
                int32 n;

                do {
                        if (get_pid_state(n) == State.Waiting) {
                                count++;
                        }
                }
                while (++n < 1000);

                return count;
        }

        // cards
        enum suit { club, diamonds, hearts, spades }
        enum value { two, three, four, five, six, seven, eight, nine, ten, jack,␣
↪queen, king, ace }
        struct card {
                value v;
                suit s;
        }

        card card1 = card(value.two, suit.club);
        card card2 = card({s: suit.club, v: value.two});

        // This function does a lot of copying
        function set_card1(card c) public returns (card previous) {
                previous = card1;
                card1 = c;
        }

        /// return the ace of spades
        function ace_of_spaces() public pure returns (card) {
                return card({s: suit.spades, v: value.ace });
        }

        /// score card
        function score_card(card c) public pure returns (uint32 score) {
                if (c.s == suit.hearts) {
                        if (c.v == value.ace) {
                                score = 14;
                        }
                        if (c.v == value.king) {
```

(continues on next page)

(continued from previous page)

```
                                score = 13;
                        }
                        if (c.v == value.queen) {
                                score = 12;
                        }
                        if (c.v == value.jack) {
                                score = 11;
                        }
                }
                // all others score 0
        }
}
```

CHAPTER 7

Contributing

Solang is in active development, so there are many ways in which you can contribute.

7.1 Target Specific

Solang supports Substrate, ewasm, and Solana. These targets need testing via integration tests. New targets like Fabric need to be added, and tests added.

7.2 How to report issues

Please report issues to github issues.

7.3 Debugging issues with LLVM

The Solang compiler builds LLVM IR. This is done via the inkwell crate, which is a "safe" rust wrapper. However, it is easy to construct IR which is invalid. When this happens you might get segfaults deep in llvm. There are two ways to help when this happens.

7.3.1 Build LLVM with Assertions Enabled

If you are using llvm provided by your distribution, llvm will not be build with `LLVM_ENABLE_ASSERTIONS=On`. See *Building LLVM from source* how to build your own.

7.3.2 Verify the IR with llc

Some issues with the IR will not be detected even with LLVM Assertions on. These includes issues like instructions in a basic block after a branch instruction (i.e. unreachable instructions).

Run `solang --emit llvm -v foo.sol` and you will get a foo.ll file, assuming that the contract in foo.sol is called foo. Try to compile this with `llc foo.ll`. If IR is not valid, llc will tell you.

7.4 Style guide

Solang follows default rustfmt, and clippy. Any clippy warnings need to be fixed. Outside of the tests, code should ideally be written in a such a way that no `#[allow(clippy::foo)]` are needed.

For test code, this is much less strict. It is much more important that tests are written, and that they have good coverage rather than worrying about clippy warnings. Feel free to sprinkle some `#[allow(clippy::foo)]` around your test code to make your merge request pass.