Solang Solidity Compiler

Release v0.3.3

Sean Young <sean@mess.org>, Cyrill Leutwiler <bigcyrill@hotma

Oct 24, 2023

USING SOLANG

Welcome to the Solang Solidity Compiler. Using Solang, you can compile smart contracts written in Solidity for Solana and Polkadot. It uses the llvm compiler framework to produce WebAssembly (WASM) or Solana SBF contract code. As result, the output is highly optimized, which saves you in gas costs or compute units.

Solang aims for source file compatibility with the Ethereum EVM Solidity compiler, version 0.8. Where differences exist, this is noted in the language documentation. The source code repository can be found on github and we have solang channels on Hyperledger Discord.

CHAPTER

ONE

CONTENTS

1.1 Installing Solang

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

1. *Download from Homebrew* (MacOS only)
2. *Download binaries*
3. *Download from a Docker container*
4. *Build using Dockerfile*
5. *Build from source*

1.1.1 Option 1: Download from Brew

Solang is available on Brew via a private tap. This works only for MacOS systems, both Intel and Apple Silicon. To install Solang via Brew, run the following command:

```
brew install hyperledger/solang/solang
```

1.1.2 Option 2: Download binaries

There are binaries available on github releases:

- Linux x86-64
- Linux arm64
- Windows x64
- MacOS intel
- MacOS arm

Download the file and save it somewhere in your `$PATH`, for example the bin directory in your home directory. If the path you use is not already in `$PATH`, then you need to add it yourself.

On MacOS, remember to give execution permission to the file and remove it from quarantine by executing the following commands:

```
chmod +x solang-mac-arm
xattr -d com.apple.quarantine solang-mac-arm
```

If you are using an Intel based Mac, please, exchange `solang-mac-arm` by `solang-mac-intel` in both of the above commands.

On Linux, permission to execute the binary is also necessary, so, please, run `chmod +x solang-linux-x86-64`. If you are using an Arm based Linux, the command is the following: `chmod +x solang-linux-arm64`.

1.1.3 Option 3: Use ghcr.io/hyperledger/solang containers

New images are automatically made available on solang containers. There is a release *v0.3.3* tag and a *latest* tag:

```
docker pull ghcr.io/hyperledger/solang:latest
```

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.

1.1.4 Option 4: Build Solang using Dockerfile

First clone the git repo using:

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

Then you can build the image using:

```
docker image build .
```

1.1.5 Option 5: Build Solang from source

In order to build Solang from source, you will need:

- Rust version 1.72.0 or higher
- A C++ compiler with support for C++17
- A build of LLVM based on the Solana LLVM tree. There are a few LLVM patches required that 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. Compatible C++17 compilers are recent releases of `gcc`, `clang` and Visual Studio 2019 or later.

To install Solang from sources, do the following:

1. *Install LLVM* from Solana's LLVM fork.
2. *Build Solang* from its source files.

Solang is also available on *crates.io*, so after completing step #1 from above, it is possible to *build it using the release* on crates.

Step 1: Install the LLVM Libraries

Solang needs a build of LLVM with some extra patches. These patches make it possible to generate code for Solana, and fixes concurrency issues in the lld linker.

You can either download the pre-built libraries from github or *build your own from source*. After that, you need to add the `bin` of your LLVM directory to your path, so that the build system of Solang can find the correct version of LLVM to use.

Linux

A pre-built version of LLVM, specifically configured for Solang, is available at https://github.com/hyperledger/solang-llvm/releases/download/llvm15-2/llvm15.0-linux-x86-64.tar.xz for x86 processors and at https://github.com/hyperledger/solang-llvm/releases/download/llvm15-2/llvm15.0-linux-arm64.tar.xz for ARM. After downloading, untar the file in a terminal and add it to your path.

```
tar Jxf llvm15.0-linux-x86-64.tar.xz
export PATH=$(pwd)/llvm15.0/bin:$PATH
```

Windows

A pre-built version of LLVM, specifically configured for Solang, is available at https://github.com/hyperledger/solang-llvm/releases/download/llvm15-2/llvm15.0-win.zip.

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

```
set PATH=%PATH%;C:\llvm15.0\bin
```

Mac

A pre-built version of LLVM for intel macs, is available at https://github.com/hyperledger/solang-llvm/releases/download/llvm15-2/llvm15.0-mac-intel.tar.xz and for arm macs there is https://github.com/hyperledger/solang-llvm/releases/download/llvm15-2/llvm15.0-mac-arm.tar.xz. After downloading, untar the file in a terminal and add it to your path like so:

```
tar Jxf llvm15.0-mac-arm.tar.xz
xattr -rd com.apple.quarantine llvm15.0
export PATH=$(pwd)/llvm15.0/bin:$PATH
```

Building LLVM from source

The LLVM project itself has a guide to installing from source which you may need to consult. Ninja is necessary for building LLVM from source. First of all, clone our LLVM repository:

```
git clone --depth 1 --branch solana-rustc/15.0-2022-12-07 https://github.com/solana-labs/
↪llvm-project
cd llvm-project
```

Now run cmake to create the makefiles. Replace the *installdir* argument to `CMAKE_INSTALL_PREFIX` 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 \
    -DLLVM_ENABLE_ZSTD=Off \
    -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
```

Step 2: Build Solang

Once you have the correct LLVM version in your path, ensure you have GNU make installed and simply run:

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

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

Alternative step 2: Build Solang from crates.io

The latest Solang release is on crates.io. Once you have the correct LLVM version in your path, ensure you have GNU make installed and simply run:

```
cargo install solang
```

1.2 Using Solang on the command line

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 WebAssembly or Solana SBF file which is ready for deployment on a chain, and an metadata file (also known as the abi).

The following targets are supported right now: Solana and Polkadot (via the `contracts` pallet runtime).

Solang supports auto-completion for multiple shells. Use `solang shell-complete --help` to learn whether your favorite shell is supported. If so, evaluate the output of `solang shell-complete <SHELL>` in order to activate it. Example installation with `bash`:

```
echo 'source <(solang shell-complete bash)' >> ~/.bashrc
```

1.2.1 Compiler Usage

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

This means that the command line is `solang compile` 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 `solana` or `polkadot`. The target must be specified.

--address-length ***length-in-bytes***
Change the default address length on Polkadot. By default, Substate uses an address type of 32 bytes. This option is ignored for any other target.

--value-length ***length-in-bytes***
Change the default value length on Polkadot. By default, Substate uses an value type of 16 bytes. This option is ignored for any other target.

-o, --output ***directory***
Sets the directory where the output should be saved. This defaults to the current working directory if not set.

--output-meta ***directory***
Sets the directory where metadata should be saved. For Solana, the metadata is the Anchor IDL file, and, for Polkadot, the .contract file. If this option is not set, the directory specified by `--output` is used, and if that is not set either, the current working directory is used.

--contract ***contract-name*** **[,** ***contract-name*****]...**
Only compile the code for the specified contracts. If any those contracts cannot be found, produce an error.

-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 working directory. This option can be specified multiple times and the directories will be searched in the order specified.

--importmap ***map=directory***
When resolving `import` directives, if the first part of the path matches *map*, search the directory provided for the file. This option can be specified multiple times with different values for map.

--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.

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

Phase:

ast-dot
Output Abstract Syntax Tree as a graphviz dot file. This can be viewed with xdot or any other tool that can visualize graphviz dot files.

cfg
Output control flow graph.

llvm-ir
Output llvm IR as text.

llvm-bc
Output llvm bitcode as binary file.

asm
Output assembly text file.

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

--no-constant-folding
Disable the *Constant Folding Pass* codegen optimization

--no-strength-reduce
Disable the *Strength Reduction Pass* codegen optimization

--no-dead-storage
Disable the *Dead Storage pass* optimization

--no-vector-to-slice
Disable the *Vector to Slice Pass* optimization

--no-cse
Disable the *Common Subexpression Elimination* optimization

--no-log-api-return-codes
Disable the *Log runtime API call results* debugging feature

--no-log-runtime-errors
Disable the *Log Runtime Errors* debugging feature

--no-prints
Disable the *Print Function* debugging feature

--release
Disable all debugging features for *Release builds:*

--config-file
Read compiler configurations from a `.toml` file. The minimal fields required in the configuration file are:

```
[package]
input_files = ["flipper.sol"]  # Solidity files to compile

[target]
name = "solana"  # Target name
```

Fields not explicitly present in the .toml acquire the compiler's default value. If any other argument is provided in the command line, for example, `solang compile --config-file --target polkadot`, the argument will be overridden. The priority for the args is given as follows: 1. Command line 2. Configuration file 3. Default values. The default name for the toml file is "solang.toml". If two configuration files exist in the same directory, priority will be given to the one passed explicitly to this argument.

--wasm-opt
wasm-opt passes for Wasm targets (0, 1, 2, 3, 4, s or z; see the wasm-opt help for more details).

--contract-authors

Specify authors for all contracts. If a *@author* tag is present, it will override this argument for the targeted contract. For specifying multiple authors, use this format: *–contract-authors author1,author2,..*

Note: This will only affect the metadata in case of substrate target.

--version
Specify contracts version. According to semver, a normal version number must take the form X.Y.Z where X, Y, and Z are non-negative integers, and must not contain leading zeroes.

Warning: If multiple Solidity source files define the same contract name, you will get a single compiled contract file for this contract name. As a result, you will only have a single contract with the duplicate name without knowing from which Solidity file it originated. Solang will not give a warning about this problem.

1.2.2 Starting a new project

solang new --target solana my_project

A solang project is a directory in which there are one or more solidity files and a `solang.toml` file where the compilation options are defined. Given these two components, a user can run `solang compile` in a similar fashion as `cargo build`.

The `solang new` command creates a new solang project with an example flipper contract, and a default `solang.toml` configuration file.

1.2.3 Generating Documentation Usage

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.

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

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

Options:

--target ***target***
This takes one argument, which can either be `solana` or `polkadot`. The target must be specified.

--address-length ***length-in-bytes***
Change the default address length on Polkadot. By default, Substate uses an address type of 32 bytes. This option is ignored for any other target.

--value-length ***length-in-bytes***
Change the default value length on Polkadot. By default, Substate uses an value type of 16 bytes. This option is ignored for any other target.

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

--importmap ***map=directory***
When resolving `import` directives, if the first part of the path matches *map*, search the directory provided for the file. This option can be specified multiple times with different values for map.

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

1.2.4 Generate Solidity interface from IDL

This command converts Anchor IDL into Solidity import files, so they can be used to call Anchor Programs from Solidity.

solang idl [–output DIR] [IDLFILE]…

For each idl file provided, a Solidity file is written. See *Calling Anchor Programs from Solidity* for an example of how to use this.

Note: There is only supported on Solana.

1.2.5 Running Solang using a container

First pull the last Solang container from solang containers:

```
docker image pull ghcr.io/hyperledger/solang
```

And if you are using podman:

```
podman image pull ghcr.io/hyperledger/solang
```

Now you can run Solang like so:

```
docker run --rm -it ghcr.io/hyperledger/solang --version
```

Or podman:

```
podman container run --rm -it ghcr.io/hyperledger/solang --version
```

If you want to compile some Solidity files, the source files need to be available inside the container. You can do this via the `-v` docker 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 ghcr.io/hyperledger/solang compile -o /
↪sources /sources/flipper.sol
```

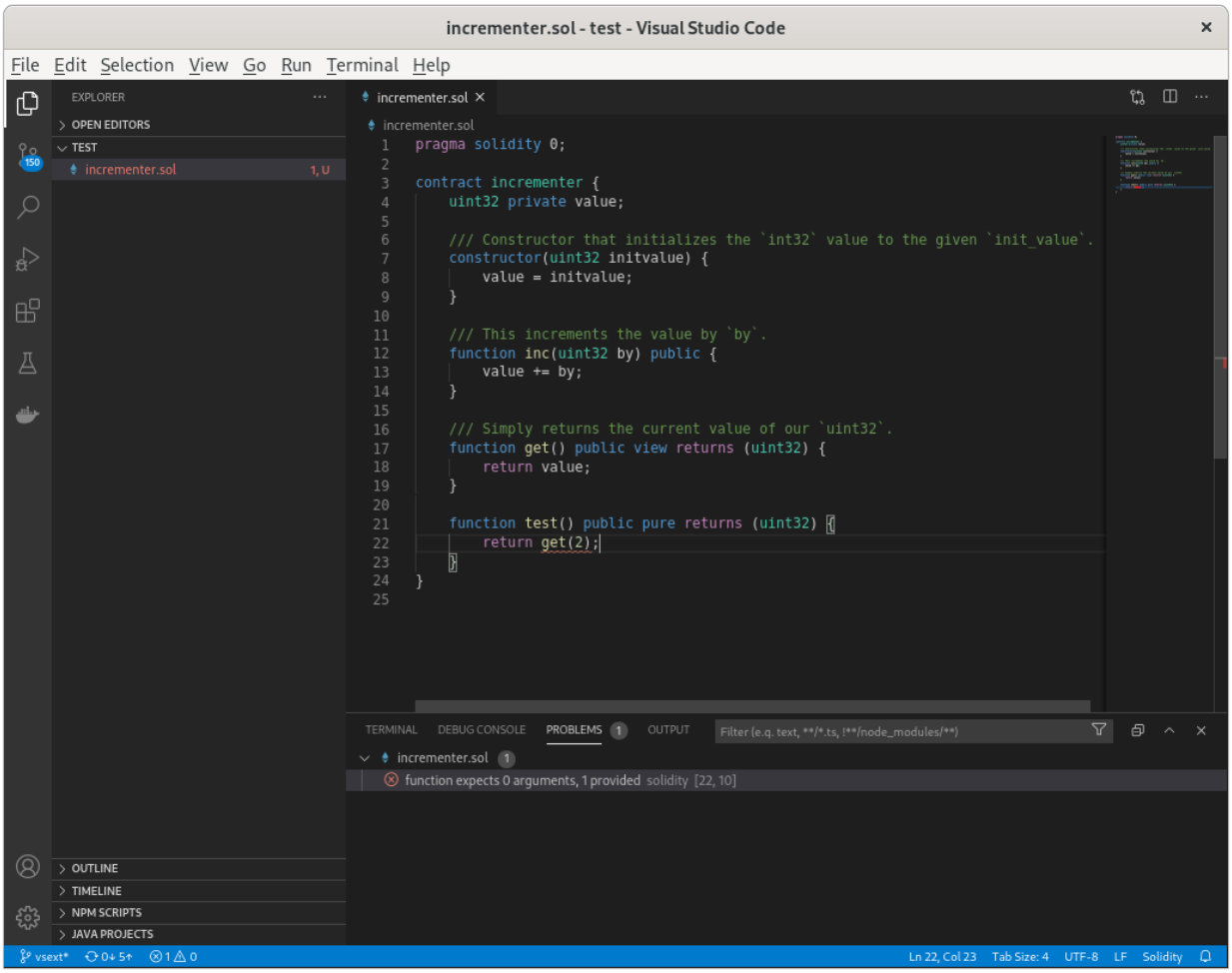
On Windows, you need to specify absolute paths:

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

1.3 Visual Studio Code Extension

Solang has a language server built into the executable, which can be used by the Visual Studio Code extension, or by any editor that can use a lsp language server. The Visual Studio Code extension provides the following:

1. Syntax highlighting.
2. Compiler warnings and errors displayed in the problems tab and marked with squiggly lines.
3. Additional information when hovering over variables, types, functions, etc. 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.

1.3.1 Solidity Language flavour

The Solidity language flavour depends on what target you are compiling for, see *Brief Language status* for a brief overview.

You can choose between the following targets:

solana
: Solidity for Solana

polkadot
: Solidity for Polkadot (Substrate contracts pallet)

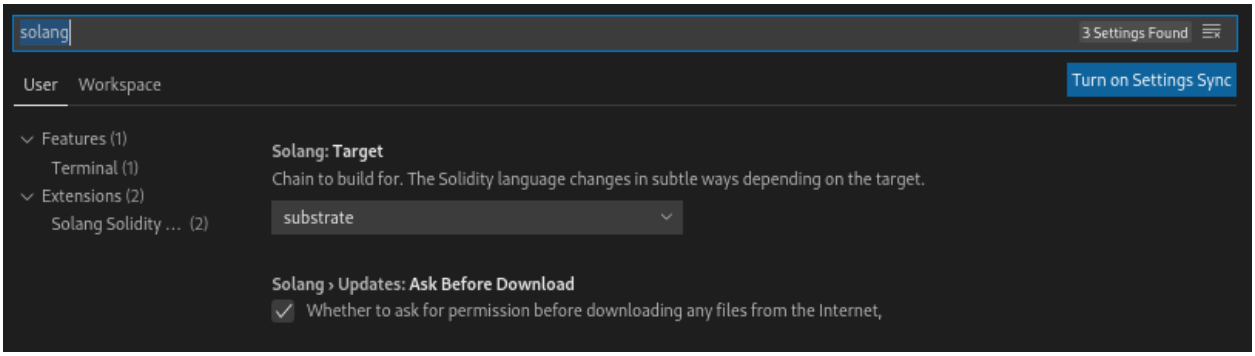
evm
: Solidity for any EVM based chain like Ethereum

Note that the language server has support for EVM, but Hyperledger Solang does not support compiling for EVM.

1.3.2 Using the extension

The extension can be found on the Visual Studio Marketplace.

When started for the first time, the extension will download the Solang binary. Once this is done, it should just automatically work. Updates are downloaded when made available. However, you should set the blockchain target in the extension settings.



1.3.3 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 a subcommand `language-server`, which starts 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/solang
cd solang/vscode
npm install
npm install -g vsce
vsce package
```

You should now have an extension file called solang-0.3.0.vsix which can be installed using `code --install-extension solang-0.3.0.vsix`.

Alternatively, the extension is 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.

1.4 Solang Solidity Examples

Here are two examples of Solidity contracts.

1.4.1 General examples

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;
        }
}
```

Example

A few simple arithmetic functions.

```
// example.sol
contract 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;
        }
```

(continues on next page)

(continued from previous page)

```
        /// Convert fahrenheit to celcius
        function fahrenheit2celcius(int32 fahrenheit) pure public returns (int32) {
                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(uint64 _pid) pure private returns (State) {
                uint64 n = 8;
                for (uint16 i = 1; i < 10; ++i) {
                        if ((i % 3) == 0) {
                                n *= _pid / uint64(i);
                        } else {
                                n /= 3;
                        }
                }

                return State(n % uint64(State.StateCount));
```

(continues on next page)

(continued from previous page)

```
        }

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

        function reap_processes() public {
                uint32 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.
                uint32 n=0;

                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 memory c) public returns (card memory previous) {
                previous = card1;
                card1 = c;
        }
```

(continues on next page)

(continued from previous page)

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

        /// score card
        function score_card(card memory c) public pure returns (uint32 score) {
                if (c.s == suit.hearts) {
                        if (c.v == value.ace) {
                                score = 14;
                        }
                        if (c.v == value.king) {
                                score = 13;
                        }
                        if (c.v == value.queen) {
                                score = 12;
                        }
                        if (c.v == value.jack) {
                                score = 11;
                        }
                }
                // all others score 0
        }
}
```

1.4.2 Solana examples

NFT example

There is an example on Solana's integration tests for a Solidity contract that manages an NFT. The contract is supposed to be the NFT itself. It can mint itself and transfer ownership. It also stores on chain information about itself, such as its URI. Please, check simple_collectible.sol for the Solidity contract and simple_collectible.spec.ts for the Typescript code that interacts with Solidity.

PDA Hash Table

On Solana, it is possible to create a hash table on chain with program derived addresses (PDA). This is done by using the intended key as the seed for finding the PDA. There is an example of how one can achieve so in our integration tests. Please, check UserStats.sol for the Solidity contract and user_stats.spec.ts for the client code, which contains most of the explanations about how the table works. This example was inspired by Anchor's PDA hash table.

1.5 Solana

1.5.1 Solana Overview

As the underlying Solana environment is different than that of Ethereum, Solidity inner workings have been modified to function properly. For example, A Solidity contract on Solana utilizes two accounts: a data account and a program account. The program account stores the contract's executable binary and owns the data account, which holds all the storage variables. On Ethereum a single account can store executable code and data.

Contract upgrades

Provided that the data layout from a new contract is compatible with that of an old one, it is possible to update the binary in the program account and retain the same data, rendering contract upgrades implemented in Solidity unnecessary. Solana's CLI tool provides a command to both do an initial deploy of a program, and redeploy it later.:

```
solana program deploy --program-id <KEYPAIR_FILEPATH> <PROGRAM_FILEPATH>
```

where <KEYPAIR_FILEPATH> is the program's keypair json file and <PROGRAM_FILEPATH> is the program binary `.so` file. For more information about redeploying a program, check Solana's documentation.

Data types

- An account address consists of a 32-bytes key, which is represented by the `address` type. This data model differs from Ethereum 20-bytes addresses.
- Solana's virtual machine registers are 64-bit wide, so 64-bit integers `uint64` and `int64` are preferable over `uint256` and `int256`. An operation with types wider than 64-bits is split into multiple operations, making it slower and consuming more compute units. This is the case, for instance, with multiplication, division and modulo using *uint256*.
- Likewise, all balances and values on Solana are 64-bit wide, so the builtin functions for *address* `.balance`, `.transfer()` and `.send()` use 64-bit integers.
- An address literal has to be specified using the `address"36VtvSbE6jVGGQytYWSaDPG7uZphaxEjpJHUUpuUbq4D"` syntax.
- Ethereum syntax for addresses `0xE0f5206BBD039e7b0592d8918820024e2a7437b9` is not supported.

Runtime

- The Solana target requires Solana v1.8.1.
- Function selectors are eight bytes wide and known as *discriminators*.
- Solana provides different builtins, e.g. `block.slot` and `tx.accounts`.
- When calling an external function or invoking a contract's constructor, one *needs to provide* the necessary accounts for the transaction.
- The keyword `this` returns the contract's program account, also know as program id.
- Contracts *cannot be types* on Solana and *calls to contracts* follow a different syntax.
- Accounts can be declared on functions using *annotations*.

Compute budget

On Ethereum, when calling a smart contract function, one needs to specify the amount of gas the operation is allowed to use. Gas serves to pay for a contract execution on chain and can be a way for giving a contract priority execution when extra gas is offered in a transaction. Each EVM instruction has an associated gas value, which translates to real ETH cost. Provided that one can afford all the gas expenses, there is no upper boundary for the amount of gas limit one can provide in a transaction, so Solidity for Ethereum has gas builtins, like `gasleft`, `block.gaslimit`, `tx.gasprice` or the Yul `gas()` builtin, which returns the amount of gas left for execution.

On the other hand, Solana is optimized for low latency and high transaction throughput and has an equivalent concept to gas: compute unit. Every smart contract function is allowed the same quantity of compute units (currently that value is 200k), and every instruction of a contract consumes exactly one compute unit. There is no need to provide an amount of compute units for a transaction and they are not charged, except when one wants priority execution on chain, in which case one would pay per compute unit consumed. Therefore, functions for gas are not available on Solidity for Solana.

Solidity for Solana incompatibilities with Solidity for Ethereum

- `msg.sender` is *not available on Solana*.
- There is no `ecrecover()` builtin function because Solana does not use the ECDSA algorithm, but there is a `signatureVerify()` function, which can check ed25519 signatures. As a consequence, it is not possible to recover a signer from a signature.
- Try-catch statements do not work on Solana. If any external call or contract creation fails, the runtime will halt execution and revert the entire transaction.
- Error definitions and reverts with error messages are not yet working for Solana.
- Value transfer with function call *does not work*.
- Many Yul builtins are not available, as specified in the *availability table*.
- External calls on Solana require that accounts be specified, as in *this example*.
- The ERC-20 interface is not compatible with Solana at the moment.

1.5.2 Getting started on Solana

Please follow the Solang Getting Started Guide.

For more examples, see the solang's integration tests.

1.5.3 Using the Anchor client library

Some notes on using the anchor javascript npm library.

- Solidity function names are converted to camelCase. This means that if in Solidity a function is called `foo_bar()`, you must write `fooBar()` in your javascript.
- Anchor only allows you to call `.view()` on Solidity functions which are declared `view` or `pure`.
- Named return values in Solidity are also converted to camelCase. Unnamed returned are given the name `return0`, `return1`, etc, depending on the position in the returns values.
- Only return values from `view` and `pure` functions can be decoded. Return values from other functions and are not accessible. This is a limitation in the Anchor library. Possibly this can be fixed.

- In the case of an error, no return data is decoded. This means that the reason provided in `revert('reason');` is not available as a return value.
- Number arguments for functions are expressed as `BN` values and not plain javascript `Number` or `BigInt`.

1.5.4 Calling Anchor Programs from Solidity

It is possible to call Anchor Programs from Solidity. You first have to generate a Solidity interface file from the IDL file using the *Generate Solidity interface from IDL*. Then, import the Solidity file in your Solidity using the `import "...";` syntax. Say you have an anchor program called `bobcat` with a function `pounce`, you can call it like so:

```
import "./bobcat.sol";
import "solana";

contract example {
    function test(address a, address b) public {
        // The list of accounts to pass into the Anchor program must be passed
        // as an array of AccountMeta with the correct writable/signer flags set
        AccountMeta[2] am = [
            AccountMeta({pubkey: a, is_writable: true, is_signer: false}),
            AccountMeta({pubkey: b, is_writable: false, is_signer: false})
        ];

        // Any return values are decoded automatically
        int64 res = bobcat.pounce{accounts: am}();
    }
}
```

1.5.5 Setting the program_id for a contract

When developing contracts for Solana, programs are usually deployed to a well known account. The account can be specified in the source code using an annotation `@program_id` if it is known beforehand. If you want to call a contract via an external call, either the contract must have a `@program_id` annotation or the `{program_id: ..}` call argument must be present.

```
@program_id("Foo5mMfYo5RhRcWa4NZ2bwFn4Kdhe8rNK5jchxsKrivA")
contract Foo {
    function say_hello() public pure {
        print("Hello from foo");
    }
}

contract OtherFoo {
    function say_bye() public pure {
        print("Bye from other foo");
    }
}

contract Bar {
    function create_foo() external {
        Foo.new();
    }
```

(continues on next page)

(continued from previous page)

```
    function call_foo() public {
        Foo.say_hello{accounts: []}();
    }

    function foo_at_another_address(address other_foo_id) external {
        OtherFoo.new{program_id: other_foo_id}();
        OtherFoo.say_bye{program_id: other_foo_id}();
    }
}
```

Note: The program_id `Foo5mMfYo5RhRcWa4NZ2bwFn4Kdhe8rNK5jchxsKrivA` was generated using the command line:

```
solana-keygen grind --starts-with Foo:1
```

1.5.6 Setting the payer, seeds, bump, and space for a contract

When a contract is instantiated, there are two accounts required: the program account to hold the executable code and the data account to save the state variables of the contract. The program account is deployed once and can be reused for updating the contract. When each Solidity contract is instantiated (also known as deployed), the data account has to be created. This can be done by the client-side code, and then the created blank account is passed to the transaction that runs the constructor code.

Alternatively, the data account can be created by the constructor, on chain. When this method is used, some parameters must be specified for the account using annotations. Annotations placed above a constructor can only contain literals or constant expressions, as is the case for first `@seed` and `@space` in the following example. Annotations can also refer to constructor arguments when placed next to them, as the second `@seed` and the `@bump` examples below. The `@payer` annotation is a special annotation that *declares an account*.

If the contract has no constructor, annotations can be paired with an empty constructor.

```
@program_id("Foo5mMfYo5RhRcWa4NZ2bwFn4Kdhe8rNK5jchxsKrivA")
contract Foo {

    @space(500 + 12)
    @seed("Foo")
    @payer(payer)
    constructor(@seed bytes seed_val, @bump bytes1 bump_val) {
        // ...
    }
}
```

Creating an account needs a payer, so at a minimum the `@payer` annotation must be specified. If it is missing, then the data account must be created client-side. The `@payer` annotation *declares a Solana account* that must be passed in the transaction.

The size of the data account can be specified with `@space`. This is a `uint64` expression which can either be a constant or use one of the constructor arguments. The `@space` should at least be the size given when you run `solang -v`:

```
$ solang compile --target solana -v examples/solana/flipper.sol
...
info: contract flipper uses at least 17 bytes account data
...
```

If the data account is going to be a program derived address, then the seeds and bump have to be provided. There can be multiple seeds, and an optional single bump. If the bump is not provided, then the seeds must not create an account that falls on the curve. When placed above the constructor, the `@seed` can be a string literal, or a hex string with the format `hex"4142"`. If before an argument, the seed annotation must refer to an argument of type `bytes`, `address`, or fixed length byte array of `bytesN`. The `@bump` must a single byte of type `bytes1`.

1.5.7 Transferring native value with a function call

The Solidity language on Ethereum allows value transfers with an external call or constructor, using the `auction.bid{value: 501}()` syntax. Solana Cross Program Invocation (CPI) does not support this, which means that:

- Specifying `value:` on an external call or constructor is not permitted
- The `payable` keyword has no effect
- `msg.value` is not supported

Note: A naive way to implement this is to let the caller transfer native balance and then inform the callee about the amount transferred by specifying this in the instruction data. However, it would be trivial to forge such an operation.

1.5.8 Receive function

In Solidity the `receive()` function, when defined, is called whenever the native balance for an account gets credited, for example through a contract calling `account.transfer(value);`. On Solana, there is no method that implements this. The balance of an account can be credited without any code being executed.

`receive()` functions are not permitted on the Solana target.

1.5.9 `msg.sender` not available on Solana

On Ethereum, `msg.sender` is used to identify either the account that submitted the transaction, or the caller when one contract calls another. On Ethereum, each contract execution can only use a single account, which provides the code and data. On Solana, each contract execution uses many accounts. Consider a rust contract which calls a Solidity contract: the rust contract can access a few data accounts, and which of those would be considered the caller? So in many cases there is not a single account which can be identified as a caller. In addition to that, the Solana VM has no mechanism for fetching the caller accounts. This means there is no way to implement `msg.sender`.

The way to implement this on Solana is to have an authority account for the contract that must be a signer for the transaction (note that on Solana there can be many signers too). This is a common construct on Solana contracts.

```
import 'solana';

contract AuthorityExample {
    address authority;
    uint64 counter;
```

(continues on next page)

(continued from previous page)

```
    constructor(address initial_authority) {
        authority = initial_authority;
    }

    @signer(authorityAccount)
    function set_new_authority(address new_authority) external {
        assert(tx.accounts.authorityAccount.key == authority && tx.accounts.
↪authorityAccount.is_signer);
        authority = new_authority;
    }

    @signer(authorityAccount)
    function inc() external {
        assert(tx.accounts.authorityAccount.key == authority && tx.accounts.
↪authorityAccount.is_signer);
        counter += 1;
    }

    function get() public view returns (uint64) {
        return counter;
    }
}
```

1.5.10 Builtin Imports

Some builtin functionality is only available after importing. The following structs can be imported via the special builtin import file `solana`.

```
import {AccountMeta, AccountInfo} from 'solana';
```

Note that `{AccountMeta, AccountInfo}` can be omitted, renamed or imported via import object.

```
// Now AccountMeta will be known as AM
import {AccountMeta as AM} from 'solana';

// Now AccountMeta will be available as solana.AccountMeta
import 'solana' as solana;
```

Note: The import file `solana` is only available when compiling for the Solana target.

Builtin AccountInfo

The account info of all the accounts passed into the transaction. `AccountInfo` is a builtin structure with the following fields:

address key
: The address (or public key) of the account

uint64 lamports
: The lamports of the accounts. This field can be modified, however the lamports need to be balanced for all accounts by the end of the transaction.

bytes data
: The account data. This field can be modified, but use with caution.

address owner
: The program that owns this account

uint64 rent_epoch
: The next epoch when rent is due.

bool is_signer
: Did this account sign the transaction

bool is_writable
: Is this account writable in this transaction

bool executable
: Is this account a program

Builtin AccountMeta

When doing an external call (aka CPI), `AccountMeta` specifies which accounts should be passed to the callee.

address pubkey
: The address (or public key) of the account

bool is_writable
: Can the callee write to this account

bool is_signer
: Can the callee assume this account signed the transaction

Builtin create_program_address

This function returns the program derived address for a program address and the provided seeds. See the Solana documentation on program derived addresses.

```
import {create_program_address} from 'solana';

contract pda {
    address token = address"TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA";

    function create_pda(bytes seed2) public returns (address) {
        return create_program_address(["kabang", seed2], token);
    }
}
```

Builtin try_find_program_address

This function returns the program derived address for a program address and the provided seeds, along with a seed bump. See the Solana documentation on program derived addresses.

```
import {try_find_program_address} from 'solana';

contract pda {
    address token = address"TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA";

    function create_pda(bytes seed2) public returns (address, bytes1) {
        return try_find_program_address(["kabang", seed2], token);
    }
}
```

1.5.11 Solana Library

In Solang's Github repository, there is a directory called `solana-library`. It contains libraries for Solidity contracts to interact with Solana specific instructions. We provide two libraries: one for SPL tokens and another for Solana's system instructions. In order to use those functionalities, copy the correspondent library file to your project and import it.

SPL-token

spl-token is the Solana native way of creating tokens, minting, burning and transferring token. This is the Solana equivalent of ERC-20 and ERC-721. Solang's repository contains a library `SplToken` to use spl-token from Solidity. The file spl_token.sol should be copied into your source tree, and then imported in your solidity files where it is required. The `SplToken` library has doc comments explaining how it should be used.

There is an example in our integration tests of how this should be used. See token.sol and token.spec.ts.

System Instructions

Solana's system instructions enable developers to interact with Solana's System Program. There are functions to create new accounts, allocate account data, assign accounts to owning programs, transfer lamports from System Program owned accounts and pay transaction fees. More information about the functions offered can be found both on Solana documentation and on Solang's system_instruction.sol file.

The usage of system instructions needs the correct setting of writable and signer accounts when interacting with Solidity contracts on chain. Examples are available on Solang's integration tests. See system_instruction_example.sol and system_instruction.spec.ts

Minimum balance

In order to instantiate a contract, you need the minimum balance required for a Solana account of a given size. There is a function `minimum_balance(uint64 space)` defined in minimum_balance.sol to calculate this.

1.5.12 Solana Account Management

In a contract constructor, one can optionally write the `@payer` annotation, which receives a character sequence as an argument. This annotation defines a Solana account that is going to pay for the initialization of the contract's data account. The syntax `@payer(my_account)` declares an account named `my_account`, which will be required for every call to the constructor.

Similarly, for external functions in a contract, one can declare the necessary accounts, using function annotations. `@account(myAcc)` declares a read only account `myAcc`, while `@mutableAccount(otherAcc)` declares a mutable account `otherAcc`. For signer accounts, the annotations follow the syntax `@signer(mySigner)` and `@mutableSigner(myOtherSigner)`.

Accessing accounts' data

Accounts declared on a constructor using the `@payer` annotation are available for access inside it. Likewise, accounts declared on external functions with any of the aforementioned annotations are also available in the `tx.accounts` vector for easy access. For an account declared as `@account(funder)`, the access follows the syntax `tx.accounts.funder`, which returns the *AccountInfo builtin struct*.

```
contract Foo {
    @account(oneAccount)
    @signer(mySigner)
    @mutableAccount(otherAccount)
    @mutableSigner(otherSigner)
    function bar() external returns (uint64) {
        assert(tx.accounts.mySigner.is_signer);
        assert(tx.accounts.otherSigner.is_signer);
        assert(tx.accounts.otherSigner.is_writable);
        assert(tx.accounts.otherAccount.is_writable);

        tx.accounts.otherAccount.data[0] = 0xca;
        tx.accounts.otherSigner.data[1] = 0xfe;

        return tx.accounts.oneAccount.lamports;
    }
}
```

External calls with accounts

In any Solana cross program invocation, including constructor calls, all the accounts a transaction needs must be informed. If the `{accounts: ...}` call argument is missing from an external call, the compiler will automatically generate the `AccountMeta` array that satisfies such a requirement. Currently, that only works if the call is done in an function declared external, as shown in the example below. In any other case, the `AccountMeta` array must be manually created, following the account ordering the IDL file specifies. If a certain call does not need any accounts, an empty vector must be passed `{accounts: []}`.

The following example shows two correct ways of calling a contract. Note that the IDL for the `BeingBuilt` contract has an instruction called `new`, representing the contract's constructor, whose accounts are specified in the following

order: `dataAccount`, `payer_account`, `systemAccount`. That is the order one must follow when invoking such a constructor.

```
import 'solana';

@program_id("SoLDxXQ9GMoa15i4NavZc61XGkas2aom4aNiWT6KUER")
contract Builder {
    function build_this() external {
        // When calling a constructor from an external function, the data account for␣
↪the contract
        // 'BeingBuilt' should be passed as the 'BeingBuilt_dataAccount' in the client␣
↪code.
        BeingBuilt.new("my_seed");
    }

    function build_that(address data_account, address payer_account) public {
        // In non-external functions, developers need to manually create the account␣
↪metas array.
        // The order of the accounts must match the order from the BeingBuilt IDL file␣
↪for the "new"
        // instruction.
        AccountMeta[3] metas = [
            AccountMeta({
                pubkey: data_account,
                is_signer: true,
                is_writable: true
                }),
            AccountMeta({
                pubkey: payer_account,
                is_signer: true,
                is_writable: true
                }),
            AccountMeta({
                pubkey: address"11111111111111111111111111111111",
                is_writable: false,
                is_signer: false
                })
        ];
        BeingBuilt.new{accounts: metas}("my_seed");


        // No accounts are needed in this call, so we pass an empty vector.
        BeingBuilt.say_this{accounts: []}("It's summertime!");
    }
}


@program_id("SoLGijpEqEeXLEqa9ruh7a6Lu4wogd6rM8FNoR7e3wY")
contract BeingBuilt {
    @space(1024)
    @payer(payer_account)
    constructor(@seed bytes my_seed) {}
```

(continues on next page)

(continued from previous page)

```
    function say_this(string text) public pure {
        print(text);
    }
}
```

1.6 Polkadot

Solang works on Polkadot Parachains integrating a recent version of the `contracts` pallets. Solidity flavored for the Polkadot target has the following differences to Ethereum Solidity:

- The address type is 32 bytes, not 20 bytes. This is what Substrate calls an "account".
- An address literal has to be specified using the `address"5GBWmgdFAMqm8ZgAHGobqDqX6tjLxJhv53ygjNtaaAn3sjeZ"` syntax
- ABI encoding and decoding is done using the SCALE encoding
- Constructors can be named. Constructors with no name will be called `new` in the generated metadata.
- There is no `ecrecover()` builtin function, or any other function to recover or verify cryptographic signatures at runtime
- Only functions called via rpc may return values; when calling a function in a transaction, the return values cannot be accessed

There is a solidity example which can be found in the examples directory. Write this to flipper.sol and run:

```
solang compile --target polkadot 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 Contracts UI, as if the contract was written in ink!.

1.6.1 Builtin Imports

Some builtin functionality is only available after importing. The following types can be imported via the special import file `polkadot`.

```
import {Hash} from 'polkadot';
import {chain_extension} from 'polkadot';
```

Note that `{Hash}` can be omitted, renamed or imported via import object.

```
// Now Hash will be known as InkHash
import {Hash as InkHash} from 'polkadot';
```

Note: The import file `polkadot` is only available when compiling for the Polkadot target.

1.6.2 Call Flags

The Substrate contracts pallet knows several flags that can be used when calling other contracts.

Solang allows a `flags` call argument of type `uint32` in the `address.call()` function to set desired flags. By default (if this argument is unset), no flag will be set.

The following example shows how call flags can be used:

```
library CallFlags {
    uint32 constant FORWARD_INPUT = 1;
    uint32 constant CLONE_INPUT = 2;
    uint32 constant TAIL_CALL = 4;
    uint32 constant ALLOW_REENTRY = 8;
}

contract Reentrant {
    function reentrant_call(
        address _address,
        bytes4 selector
    ) public returns (bytes ret) {
        (bool ok, ret) = _address.call{flags: CallFlags.ALLOW_REENTRY}(selector);
        require(ok);
    }
}
```

1.6.3 Reverts and error data decoding

When a contract reverts, the returned error data is what the EVM would return. `assert()`, `require()`, or `revert()` will revert the contract execution, where the revert reason (if any) is encoded as `Error(string)` and provided in the execution output. Solidity contracts can also revert with a `Panic(uint256)` (please refer to the Ethereum Solidity language documentation for more information about when `Panic` might be returned). Uncaught exceptions from calling and instantiating contracts or transferring funds will be bubbled up back to the caller.

Note: Solidity knows about `Error`, `Panic` and custom errors. Please, also refer to the Ethereum Solidity documentation, for more information.

The metadata contains all error variants that the contract *knows* about in the `lang_error` field.

> **Warning:** Never trust the error data.
>
> Solidity contracts do bubble up uncaught errors. This can lead to situations where the contract reverts with error data unknown to the contracts. Examples of this include bubbling up custom error data from the callee or error data from an `ink!` contract.

The 4 bytes selector of the error data can be seen as the enum discriminator or index. However, because SCALE encoding does not allow index larger than 1 byte, the hex-encoded error selector is provided as the path of the error variant type in the metadata.

In the following example, the `Panic` variant of `lang_error` is of type `10`, which looks like this:

```
{
  "id": 10,
  "type": {
    "def": {
      "composite": {
        "fields": [
          {
            "type": 9
          }
        ]
      }
    },
    "path": [
      "0x4e487b71"
    ]
  }
}
```

From this follows that error data matching the `Panic` selector of *0x4e487b71* can be decoded according to type `10` (where the decoder must exclude the first 4 selector bytes).

The general process of decoding the output data of Solang Solidity contracts is as follows:

1. The compiler of the contract must be Solang (check the `compiler` field in the contract metadata).
2. If the revert flag is **not** set, the contract didn't revert and the output should be decoded as specified in the message spec.
3. If the output length is smaller than 4 bytes, the error data can't be decoded (contracts may return empty error data, for example if `revert()` without arguments is used).
4. If the first 4 bytes of the output do **not** match any of the selectors found in `lang_error`, the error can't be decoded.
5. **Skip** the selector (first 4 bytes) and decode the remaining data according to the matching type found in *lang_error*.

1.7 Brief Language status

The Solidity language supported by Solang aims to be compatible with the latest Ethereum Foundation Solidity Compiler, version 0.8 with some small exceptions.

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

As with any new project, bugs are possible. Please report any issues you may find to github.

Differences:

- libraries are always statically linked into the contract code
- Solang generates WebAssembly or Solana SBF rather than EVM.
- Packed encoded uses little endian encoding, as WASM and SBF are little endian virtual machines.

Unique features to Solang:

- Solang can target different blockchains and some features depending on the target. For example, Polkadot uses a different ABI encoding and allows constructors to be named.
- 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));`
- Ethereum style address literals like `0xE0f5206BBD039e7b0592d8918820024e2a7437b9` are not supported on Polkadot or Solana, but are supported for EVM.
- On Polkadot and Solana, base58 style encoded address literals like `address"5GBWmgdFAMqm8ZgAHGobqDqX6tjLxJhv53ygjNtaaAn3sjeZ"` are supported, but not with EVM.
- On Solana, there is special builtin import file called `'solana'` available.
- On Polkadot, there is special builtin import file called `'polkadot'` available.
- Different blockchains offer different builtins. See the *builtins documentation*.
- There are many more differences, which are noted throughout the documentation.

1.8 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 only single contract in each file, and keep contract name the same as the file name (with the *.sol* extension).

1.9 Imports

The `import` directive is used to import items from other Solidity files. This can be useful to keep a single definition in one file, which can be used in multiple other files. For example, you could have an interface in one source file, which several contracts implement or use which are in other files. Solidity imports are somewhat similar to JavaScript ES6, however there is no export statement, or default export.

The following items are always exported, which means they can be imported into another file.

- global constants
- struct definitions
- enums definitions
- event definitions
- global functions
- free standing 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 just a select few items. 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 paths specified with the `--importpath` commandline option. If the file is relative, e.g. `import "./defines.sol";` or `import "../defines.sol";`, then the directory relative to the parent file is used. 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";
```

Another method for locating files is using import maps. This maps the first directory of an import path to a different location on the file system. Say you add the command line option `--importmap @openzeppelin=/opt/openzeppelin-contracts/contracts`, then

```
import "@openzeppelin/interfaces/IERC20.sol";
```

will automatically map to */opt/openzeppelin-contracts/contracts/interfaces/IERC20.sol*.

Everything defined in *defines.sol* is now usable in your Solidity file. However, if an item with the same name is defined in *defines.sol* and also in the current file, you will get a warning. It is permitted to import the same file more than once.

It is also possible to rename an import. In this case, only item *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 items 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* as *defs.foo*. As long as there is no previous item *defs*, there can be no naming conflict.

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

There is another syntax, which does exactly the same.

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

Just like string literals, import paths can have escape sequences. This is a confusing way of writing *a.sol*:

```
import "\x61.sol";
```

It is possible to use \ Windows style path separators on Windows, but it is not recommended as they do not work on platforms other than Windows (they do not work on WSL either). Note they have to be written as \\ due to escape sequences.

1.10 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.

1.10.1 About pragma solidity versions

Ethereum Solidity checks the value of `pragma version` against the compiler version, and gives an error if they do not match. Ethereum Solidity is often revising the language in various small ways which make versions incompatible which other. So, the version pragma ensures that the compiler version matches what the author of the contract was using, and ensures the compiler will give no unexpected errors.

Solang takes a different view:

1. Solang tries to remain compatible with different versions of ethereum solidity; we cannot publish a version of solang for every version of the ethereum solidity compiler.
2. We also have compatibility issues because we target multiple blockchains, so the version would not be sufficient.
3. We think that the compiler version should not be a property of the source, but of the build environment. No other language set the compiler version in the source code.

If anything, some languages allow conditional compilation based on the compiler version, which is much more useful.

1.11 Types

The following primitive types are supported.

1.11.1 Boolean Type

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

1.11.2 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:

```
value 300 does not fit into type 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 Solana SBF 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 or compute units.

1.11.3 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 `hex"414243"`, or a text string `"ABC"`, or a hex value `0x414243`.

```
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 to aid 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);
```

1.11.4 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 EVM, an address is 20 bytes. Solana and Polkadot have an address length of 32 bytes. The format of an address literal depends on what target you are building for. On EVM, ethereum addresses can be specified with a particular hexadecimal number.

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

The hexadecimal string should be 40 hexadecimal 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'
```

Polkadot 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 should be declared `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: Polkadot can be compiled with a different type for Address. If your target runtime has a different length for address, you can specify `--address-length` on the command line.

1.11.5 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. Enum are limited to 256 values.

```
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
}

abstract 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;
```

(continues on next page)

(continued from previous page)

```
            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;
        }
    }
}
```

1.11.6 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) {
```

(continues on next page)

(continued from previous page)

```
            if (c.v == value.ace) {
                score = 14;
            }
            if (c.v == value.king) {
                score = 13;
            }
            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) {
```

(continues on next page)

(continued from previous page)

```
        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 users) public {
        // ...
    }
}

abstract contract db {
    struct users {
        user[] field1;
        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*. Notice that the *db.users* struct type is used in the function *authenticate* before it is declared. In Solidity, types can be always be used before their declaration, or even before the `import` directive.

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 and executed for each field. In the *set_card1* function, the following is done:

- 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;
```

(continues on next page)

(continued from previous page)

```
        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");
    }
}
```

1.11.7 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. For example:

```
contract primes {
    function primenumber(uint32 n) public pure returns (uint64) {
        uint64[10] primes = [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:

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

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

        set_2(val);

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

On Solang, it is not necessary to cast the first element of the array literal.

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.

On Polkadot, the SCALE encoding uses 32 bytes for both types. Similarly, the borsh encoding used on Solana uses 32 bytes for both types.

1.11.8 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.

```
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 accessing 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;
    }
    user[] 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.

1.11.9 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.

1.11.10 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);
    }
}
```

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[]`.

1.11.11 Mappings

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

- They only work as storage variables
- 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 {
        // This construction is not recommended, because it requires two hash␣
↪calculations.
        // See the tip below.
        users[name].exists = true;
        users[name].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];
    }
}
```

Mappings may have a name for the key or the value, for example: `mapping(address owner => uint64 balance)`. The names are used in the metadata of the contract. If the mapping is public, the accessor function will have named arguments and returns.

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 can be optimized like the following.

```
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;
}
```

Here the storage slot for the struct is calculated only once, avoiding another expensive keccak256 calculation.

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

Note: Solidity on Ethereum and on Polkadot takes the keccak 256 hash of the key and the storage slot, and simply uses that to find the entry. Its underlying hash table does not use separate chaining for collision resolution. The scheme is simple and avoids "hash flooding" attacks that utilize hash collisions to exploit the worst-case time complexity for a separately chained hash table. When too many collisions exist in a such a data structure, it degenerates to a linked list, whose time complexity for searches is O(n).

In order to implement mappings on Solana's storage, a new scheme must be found to prevent this attack. SipHash is a hash algorithm that solves the problem, but it cannot be used in smart contracts since there is no place to store secrets. Separate chaining for collision handling is needed since Solana accounts have a much smaller address space than the 256 bit storage slots. Any suggestions for solving this are very welcome!

SipHash may serve as a way to implement mappings in memory, which would allow them to be local variables in functions. Although, a safe alternative to random seeds still needs to be found.

1.11.12 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 on 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 {
        // Note: on Solana, new Contract() requires an address
        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);
    }
}
```

Note: On Solana, contracts cannot exist as types, so contracts cannot be function parameters, function returns or variables. Contracts on Solana are deployed to a defined address, which is often known during compile time, so there is no need to hold that address as a variable underneath a contract type.

1.11.13 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, bool) internal pure y;

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

    function func1(int32 arg, bool arg2) internal view 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.

Polkadot

```
contract ft {
    function test(paffling p) public {
```

(continues on next page)

(continued from previous page)

```
        // 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 {
    // 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");
    }
}
```

Solana

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

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

contract paffling {
    // 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{accounts: []}(1, "paffled");
    }
}
```

On Solana, external calls from variables of type external functions require the `accounts` call argument. The compiler cannot determine the accounts such a function needs, so it does not automatically generate the `AccountsMeta` array.

```
function test(function(int32, string) external myFunc) public {
    myFunc{accounts: []}(24, "accounts");
}
```

1.11.14 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().

1.11.15 User Defined Types

A user defined type is a new type which simply wraps an existing primitive type. First, a new type is declared with the `type` syntax. The name of the type can now be used anywhere where a type is used, for example in function arguments or return values.

```
type Value is uint128;

contract Foo {
    function inc_and_wrap(int128 v) public returns (Value) {
        return Value.wrap(uint128(v + 1));
    }

    function dec_and_unwrap(Value v) public returns (uint128) {
        return Value.unwrap(v) - 1;
    }
}
```

Note that the wrapped value `Value v` cannot be used in any type of arithmetic or comparision. It needs to be unwrapped before it can be used.

User Defined Types can be used with *user defined operators*.

1.12 Expressions

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

1.12.1 Arithmetic operators

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

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

Parentheses can be used too, of course:

```
uint32 celsius = (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;
```

No overflow checking is generated in *unchecked* blocks, like so:

```
contract foo {
    function f(int64 n) public {
```

(continues on next page)

(continued from previous page)

```
        unchecked {
            int64 j = n - 1;
        }
    }
}
```

1.12.2 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.

1.12.3 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*.

1.12.4 Conditional operator

The ternary conditional operator `? :` is supported:

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

1.12.5 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.

1.12.6 Increment and Decrement operators

The post-increment and pre-increment operators are implemented by reading the variable's value before or after modifying it. `i++``returns the value of ``i` before incrementing, and `++i` returns the value of `i` after incrementing.

1.12.7 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.

Polkadot

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

Solana

```
contract kadowari {
    function nomi() public {
        // Contracts are not allowed as variables on Solana
        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.

Polkadot

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

    function nokogiri(int256 a) public {
        // ...
    }
}
```

Solana

```
@program_id("H3AthiA2C1pcMahg17nEwqr9628gkXUnnzWJJ3iSDekL")
contract kadowari {
    function nomi() public {
        this.nokogiri{accounts: []}(102);
    }

    function nokogiri(int256 a) public {
        // ...
    }
}
```

Note: On Solana, `this` returns the program account. If you are looking for the data account, please use `tx.accounts.dataAccount.key`.

1.12.8 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(int256 x) public {
        if (x < type(int16).min || x > type(int16).max) {
            revert("value will not fit");
        }

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

The EIP-165 interface value can be retrieved using the syntax `type(...).interfaceId`. This is only permitted on interfaces. The interfaceId is simply an bitwise XOR of all function selectors in the interface. This makes it possible to uniquely identify an interface at runtime, which can be used to write a *supportsInterface()* function as described in the EIP.

The contract code for a contract, i.e. the binary WebAssembly or Solana SBF, can be retrieved using the `type(c).creationCode` and `type(c).runtimeCode` fields, as `bytes`. On EVM, the constructor code is in the `creationCode` and all the functions are in the `runtimeCode`. Polkadot and Solana use the same code for both, so those fields will evaluate to the same value.

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

contract other {
    function foo() public returns (bool) {
        return true;
    }
}
```

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.

1.12.9 Ether, Sol, 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
`lamports`	1
`sol`	1_000_000_000
`wei`	1
`gwei`	1_000_000_000
`ether`	1_000_000_000_000_000_000

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

1.12.10 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 sign changes by adding an explicit 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 is idiomatic to match the return value type with the argument type. 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 an array of one byte, a conversion from `byte` to `uint8` requires a cast. This is because `byte` is an alias for `bytes1`.

1.13 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 shadow the original item and generate a warning as it is no longer accessible.

```
contract test {
    uint256 foo = 102;
    uint256 bar;

    function foobar() public {
        // AVOID: this shadows the contract storage variable foo
        uint256 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:

```
contract Foo {
    function foo() public {
        // new block is introduced with { and ends with }
        {
            uint256 a;

            a = 102;
        }

        // ERROR: a is out of scope
        // uint256 b = a + 5;
    }
}
```

1.13.1 If statement

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

```
contract Foo {
    function foo(uint32 n) public {
        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.

You can optionally add an `else` block which is executed only if the condition evaluates to false.

```
contract Foo {
    function foo(uint32 n) public {
        if (n > 10) {
            // do something
        } else {
            // do something different
        }
    }
}
```

1.13.2 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 body is never executed:

```
contract Foo {
    function foo(uint256 n) public {
        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:

```
contract Foo {
    function bar(uint256 n) public returns (bool) {
        return false;
```

(continues on next page)

(continued from previous page)

```
    }

    function foo(uint256 n) public {
        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;
            }

            // only executed if both if statements were false
            print("neither true");
        }

        n = 102;
    }
}
```

1.13.3 Do While statement

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

```
contract Foo {
    function bar(uint256 n) public returns (bool) {
        return false;
    }

    function foo(uint256 n) public {
        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;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

1.13.4 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:

```
contract Foo {
    function foo() public {
        for (uint256 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):

```
contract Foo {
    function foo(uint256 n) public {
        // all three omitted
        for (;;) {
            // there must be a way out
            if (n == 0) {
                break;
            }
        }
    }
}
```

1.13.5 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 {
```

(continues on next page)

(continued from previous page)

```
        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.

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

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

The right hand side of an destructure may contain the ternary conditional operator. The number of elements in both sides of the conditional must match the left hand side of the destructure statement.

```
contract Foo {
    function test(bool cond) public {
        (int32 a, int32 b, int32 c) = cond ? (1, 2, 3) : (4, 5, 6);
    }
}
```

1.13.6 Try Catch Statement

Solidity's try-catch statement can only be used with external calls or constructor calls using `new`. The compiler will refuse to compile any other expression.

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 in the caller 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.

Warning: On Solana, any transaction that fails halts the execution of a contract. The try-catch statement, thus, is not supported for Solana contracts and the compiler will raise an error if it detects its usage.

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

    function never() public pure {}
}

contract runner {
    function test() public {
```

(continues on next page)

(continued from previous page)

```
        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 abort.abort() returns (int32 a, bool b) {
            // call succeeded; return values are in a and b
```

(continues on next page)

(continued from previous page)

```
        } catch (bytes raw) {
            // call failed with raw error in raw
        }
    }
}
```

Note: Try-catch only supports `Error` and `Panic` errors with an explicit catch clause. Calls reverting with a custom error will be caught in the catch-all clause (`catch (bytes raw)`) instead. If there is no catch-all clause, custom errors will bubble up to the caller.

1.14 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;
}
```

1.15 `Using` directive

1.15.1 Binding methods to types with `using`

Methods can be bound to builtin types and any user-defined types like structs using the `using` syntax. This can be done either using libraries or free standing functions.

`using` with free standing functions

First, declare a function with one or more arguments. Once the function is bound with `using`, it can be called like a method.

```
function mask(uint v, uint bits) returns (uint) {
    return v & ((1 << bits) - 1);
}

function odd(uint v) returns (bool) {
    return (v & 1) != 0;
}

contract c {
    using {mask, odd} for *;

    uint v;
```

(continues on next page)

(continued from previous page)

```
    function set_v(uint n) public {
        v = n.mask(16);
    }
}
```

The `using` declaration can be done on file scope. In this case, the type must be specified in place of `*`. The first argument must match the type that is be used in the `using` declaration.

If a user-defined type is used, the the `global` keyword can be used. This means the `using` binding can be used in any file, even when the type is imported.

```
struct User {
    string name;
    uint count;
}

function clear_count(User memory user) {
    user.count = 0;
}

using {clear_count} for User global;
```

Now even when `User` is imported, the clear_count() method can be used.

```
import {User} from "./user.sol";

contract c {
    function foo(User memory user) public {
        user.clear_count();
    }
}
```

User defined Operators

The `using` directive can be used to bind operators for *user defined types* to functions. A binding can be set for the operators: ==, !=, >=, >, <=, <, ~, &, |, ^, - (both negate and subtract), +, *, /, and %.

First, declare a function with the correct prototype that implements the operator.

- The function must be free standing: declared outside a contract.
- The function must have `pure` mutability.
- All the parameters must be the same user type.
- The number of arguments depends on which operator is implemented; binary operators require two and unary operators, one.
- The function must return either `bool` for the comparison operators, or the same user type as the parameters for the other operators.

Then, bind the function to the operator using the syntax `using {function-name as operator} for user-type global;`. Operators can only be defined with `global` set. Note that the `-` operator is used for two operators: subtract and negate. In order to bind the unary negate operator, the function must have a single parameter. For the subtract operator, two parameters are required.

```
type Bitmap is int256;

function sub(Bitmap a, Bitmap b) pure returns (Bitmap) {
        return Bitmap.wrap(Bitmap.unwrap(a) - Bitmap.unwrap(b));
}

function add(Bitmap a, Bitmap b) pure returns (Bitmap) {
        return Bitmap.wrap(Bitmap.unwrap(a) + Bitmap.unwrap(b));
}

function neg(Bitmap a) pure returns (Bitmap) {
        return Bitmap.wrap(-Bitmap.unwrap(a));
}

using {sub as -, neg as -, add as +} for Bitmap global;

function foo(Bitmap a, Bitmap b) {
        Bitmap c = a + b;
        // ...
}
```

`using` with libraries

A library may be used for handling methods on a type. First, declare a library with all the methods you want for a type, and then bind the library to the type with `using`.

```
struct User {
    string name;
    uint name;
}

library UserLibrary {
    function clear_count(User user) internal {
        user.count = 0;
    }

    function inc(User user) internal {
        user.count++;
    }

    function dec(User user) internal {
        require(user.count > 0);
        user.count--;
    }
}

using UserLibrary for User global;
```

Scope for `using`

The `using` declaration may be scoped in various ways:

- Globally by adding the `global` keyword. This means the methods are available in any file.
- Per file, by omitting the `global` keyword
- Per contract, by putting the `using` declaration in a contract definition

If the scope is per contract, then the type maybe be replaced with * and the type from the first argument of the function will be used.

1.16 Contracts

1.16.1 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 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;
        }
}
```

A constructor can have any number of arguments. If a constructor has arguments, they must be supplied when the contract is deployed.

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

Note: Solang allows naming constructors in the Polkadot target:

```
abstract contract Foo {
    constructor my_new_foo() {}
}
```

Constructors without a name will be called `new` in the metadata.

Note that constructor names are only used in the generated metadata. For contract instantiation, the correct constructor matching the function signature will be selected automatically.

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. While on Polkadot and Ethereum constructors return the address of the instantiated contract, on Solana, the address is either passed to the call using the `{program_id: ...}` call argument or is declared above a contract with the `@program_id` annotation. As the constructor does not return anything and its purpose is only to initialize the data account, the syntax `new Contract()``is not idiomatic on Solana. Instead, a function ``new` is made available to call the constructor.

Polkadot

```
contract hatchling {
    string name;
    address private origin;

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

    function root() public returns (address) {
        return origin;
    }
}

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

Solana

```
@program_id("5afzkvPkrshqu4onwBCsJccb1swrt4JdAjnpzK8N4BzZ")
contract hatchling {
    string name;
    address private origin;

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

    function root() public returns (address) {
        return origin;
    }
}
```

(continues on next page)

(continued from previous page)

```
contract adult {
    function test() external {
        hatchling.new("luna", address(this));
    }
}
```

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.

Note: On Solana, the *Try Catch Statement* statement is not supported, as any failure will cause the entire transaction to fail.

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:

```
contract hatchling {
    string name;
    address private origin;

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

    function root() public returns (address) {
        return origin;
    }
}

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

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

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

Note: On Solana, this functionality is not available.

Setting the salt, gas, and address for the new contract

Note: The gas or salt cannot be set on Solana. However, when creating a contract on Solana, the address of the new account must be set using `address:`.

When a new contract is created, the address for the new contract is a hash of the input (the encoded constructor arguments) to the new contract and the salt. A contract cannot be created twice with the same input and salt. By giving a different salt, the same input can be used twice for a new contract. The salt can be set using the `{salt: hex"439d399ee3b5b0fae6c8d567a8cbfa22d59f8f2c5fe308fd0a92366c116e5f1a"}` syntax, or if it is omitted, then a random value is used.

Specifying a salt will remove the need for generating a random value at runtime, however care must be taken to avoid using the same salt more than once. Creating a contract twice with the same salt and arguments will fail. The salt is of type `bytes32`.

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

```
contract hatchling {
    string name;
    address private origin;

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

    function root() public returns (address) {
        return origin;
    }
}

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

Solana constructors

Solidity contracts are coupled to a data account, which stores the contract's state variables on the blockchain. This account must be initialized before calling other contract functions, if they require one. A contract constructor initializes the data account and can be called with the `new` function. When invoking the constructor from another contract, the data account to initialize appears in the IDL file and is identified as `contractName_dataAccount`. In the example below, the IDL for the instruction `test` requires the `hatchling_dataAccount` account to be initialized as the new contract's data account.

```
contract hatchling {
    string name;

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

(continues on next page)

(continued from previous page)

```
        name = id;
    }
}

contract adult {
    function test(address id) external {
        hatchling.new{program_id: id}("luna");
    }
}
```

When there are no call arguments to a constructor call, the compiler will automatically create the `AccountMeta` array the constructor call needs. Due to the impossibility to track account ordering in private, internal and public functions, such a call argument is only allowed in functions with `external` visibility. This automatic account management only works, however, if there is a single instantiation of a particular contract type.

Alternatively, the data account to be initialized can be provided using the `accounts` call argument. In this case, one needs to instantiate a fixed length array of type `AccountMeta` to pass to the call. The array must contain all the accounts the transaction is going to need, in addition to the data account to be initialized.

For the creation of a contract, the data account must the **first** element in such a vector and the system account 11111111111111111111111111111111 must also be present. If the constructor one is calling has the *@payer annotation*, the payer account should appear in the array as well. Moreover, the `is_signer` and `is_writable` bool flags need to be properly set, according to the following example:

```
import 'solana';

contract creator {

    @mutableSigner(data_account_to_initialize)
    @mutableSigner(payer)
    function create_with_metas() external {
        AccountMeta[3] metas = [
            AccountMeta({
                pubkey: tx.accounts.data_account_to_initialize.key,
                is_signer: true,
                is_writable: true}),
            AccountMeta({
                pubkey: tx.accounts.payer.key,
                is_signer: true,
                is_writable: true}),
            AccountMeta({
                pubkey: address"11111111111111111111111111111111",
                is_writable: false,
                is_signer: false})
        ];

        Child.new{accounts: metas}();

        Child.use_metas{accounts: []}();
    }
}

@program_id("Chi1d5XD6nTAp2EyaNGqMxZzUjh6NvhXRxbGHP3D1RaT")
```

(continues on next page)

(continued from previous page)

```
contract Child {
    @payer(payer)
    constructor() {
        print("In child constructor");
    }

    function use_metas() pure public {
        print("I am using metas");
    }
}
```

The sequence of the accounts in the `AccountMeta` array matters and must follow the *IDL ordering*.

Calling a contract on Solana

A call to a contract on Solana follows a different syntax than that of Solidity on Ethereum or Polkadot. As contracts cannot be a variable, calling a contract's function follows the syntax `Contract.function()`. If the contract definition contains the `@program_id` annotation, the CPI will be directed to the address declared inside the annotation.

If that annotation is not present, the program address must be manually specified with the `{program_id: ... }` call argument. When both the annotation and the call argument are present, the compiler will forward the call to the address specified in the call argument.

```
contract Polymath {
    function call_math() external returns (uint) {
        return Math.sum(1, 2);
    }
    function call_english(address english_id) external returns (string) {
        return English.concatenate{program_id: english_id}("Hello", "world");
    }
}

@program_id("5afzkvPkrshqu4onwBCsJccb1swrt4JdAjnpzK8N4BzZ")
contract Math {
    function sum(uint a, uint b) external returns (uint) {
        return a + b;
    }
}

contract English {
    function concatenate(string a, string b) external returns (string) {
        return a + b;
    }
}
```

1.16.2 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.

Specifying base contracts

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

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

contract b {
    int256 foo;

    function func2() public {}

    constructor() {}
}

contract c {
    int256 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`.

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

contract b is c {
    int256 foo;

    function func2() public {}

    constructor() {}
}

contract c {
    int256 bar;

    constructor() {}
```

(continues on next page)

(continued from previous page)

```
    function func1() public {}
}
```

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.

```
contract a is b {
    function func(int256 a) public override returns (int256) {
        return a + 11;
    }
}

contract b {
    function func(int256 a) public virtual returns (int256) {
        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.

```
contract a is b, c {
    function func(int256 a) public override(b, c) returns (int256) {
        return a + 11;
    }
}

contract b {
    function func(int256 a) public virtual returns (int256) {
        return a + 10;
    }
}

contract c {
    function func(int256 a) public virtual returns (int256) {
        return a + 5;
    }
}
```

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

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

abstract 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();
    }

    function bar2() internal returns (uint64) {
        // this explicitly says "call foo of base contract a", and dispatch is not␣
↪virtual
        // however, if the call is written as a.foo{program_id: id_var}(), this␣
↪represents
        // an external call to contract 'a' on Solana.
        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;
    }
}

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

(continues on next page)

(continued from previous page)

```
}
```

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(b1, b2) returns (uint64) {
        return 2;
    }
}

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

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

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.

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

contract b is c(2) {
    int256 foo;

    function func2(int256 i) public {}

    constructor(int256 j) {}
}

contract c {
    int256 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.

```
contract a is b {
    constructor(int256 i) b(i + 2) {}
}

contract b is c {
    int256 foo;

    function func2() public {}

    constructor(int256 j) c(int32(j + 3)) {}
}

contract c {
    int256 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.

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 contract a {
    function func2() public virtual;
}
```

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 contract a is b {
    constructor() {}
}

contract b {
    int256 public j;
```

(continues on next page)

(continued from previous page)

```
    constructor(int256 _j) {}
}

contract c is a {
    int256 public k;

    constructor(int256 k) b(k * 2) {}
}
```

1.17 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 {
    uint256 public counter = 1;

    function hit() public {
        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`.

1.17.1 Immutable Variables

A variable can be declared *immutable*. This means that it may only be modified in a constructor, and not in any other function or modifier.

```
contract foo {
    uint256 public immutable bar;

    constructor(uint256 v) {
        bar = v;
    }

    function hit() public {
        // this is not permitted
        // bar++;
    }
}
```

This is purely a compiler syntax feature, the generated code is exactly the same.

1.17.2 Accessor Functions

Any contract storage variable which is declared public, automatically gets an accessor function. This function has the same name as the variable name. So, in the example above, the value of counter can retrieved by calling a function called `counter`, which returns `uint`.

If the type is either an array or a mapping, the key or array indices become arguments to the accessor function.

```
contract ethereum {
    // As a public mapping, this creates accessor function called balance, which takes
    // an address as an argument, and returns an uint
    mapping(address => uint256) public balances;

    // A public array takes the index as an uint argument and returns the element,
    // in this case string.
    string[] users;
}
```

The accessor function may override a method on a base contract by specifying `override`. The base function must be virtual and have the same signature as the accessor. The `override` keyword only affects the accessor function, so it can only be used in combination with public variables and cannot be used to override a variable in the base contract.

```
contract foo is bar {
    int256 public override baz;
}

contract bar {
    function baz() public virtual returns (int256) {
        return 512;
    }
}
```

1.17.3 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;
        int256[] 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;
    }
}
```

1.18 Interfaces and libraries

1.18.1 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.

```
// SPDX-License-Identifier: MIT

// Interface for an operator that performs an operation on two int32 values.
interface Operator {
    function performOperation(int32 a, int32 b) external returns (int32);
}

contract Ferqu {
    Operator public operator;

    // Constructor that takes a boolean parameter 'doAdd'.
    constructor(bool doAdd) {
        if (doAdd) {
            operator = new Adder();
        } else {
            operator = new Subtractor();
        }
    }

    // Function to calculate the result of the operation performed by the chosen␣
↪operator.
    function calculate(int32 a, int32 b) public returns (int32) {
        return operator.performOperation(a, b);
    }
}

// Contract for addition, implementing the 'Operator' interface.
contract Adder is Operator {
    function performOperation(int32 a, int32 b) public pure override returns (int32) {
        return a + b;
    }
}

// Contract for subtraction, implementing the 'Operator' interface.
contract Subtractor is Operator {
    function performOperation(int32 a, int32 b) public pure 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

1.18.2 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.

```
library InstanceLibrary {
    function getMax(uint64 a, uint64 b) external pure returns (uint64) {
        return a > b ? a : b; // If 'a' is greater than 'b', it returns 'a'; otherwise,␣
↪it returns 'b'.
    }
}

contract TestContract {
    using InstanceLibrary for uint64;

    // Calculate and return the maximum value between x and 65536 using the␣
↪InstanceLibrary.
    function calculateMax(uint64 x) public pure returns (uint64) {
        return x.getMax(65536);
    }
}
```

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, libraries are a special contract type and are called using *delegatecall*. Solang statically links the library calls into your contract code. This generates larger contract code, however it reduces the call overhead and make it possible to do compiler optimizations across library and contract code.

1.18.3 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.

```
library Library {
    function set(
        int32[100] storage data,
        uint256 index,
        int32 value
    ) internal  {
        data[index] = value;
```

(continues on next page)

(continued from previous page)

```
    }
}

contract TestContract {
    using Library for int32[100];

    int32[100] public dataArray;

    function setElement(uint256 index, int32 value) public {
        dataArray.set(index, value);
    }
}
```

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.

1.19 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.

Warning: On Solana, the `indexed` event field attribute has no effect. All event attributes will be encoded as data to be passed for Solana's `sol_log_data` system call, regardless if the `indexed` keyword is present or not. This behavior follows what Solana's Anchor framework does.

In Polkadot, 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 {
    function sign(address counter_party, uint contract_no) public {
        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(address user, string name) public {
        emit UserModified({ user: user, name: name });
    }

    function set_groupid(address user, uint64 id) public {
        emit UserModified({ user: user, 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.

1.20 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 (uint256 value) {
    value = 102;
}
```

(continues on next page)

(continued from previous page)

```
contract foo {
    uint256 bound = get_initial_bound();

    /** set bound for get with bound */
    function set_bound(uint256 _bound) public {
        bound = _bound;
    }

    // Clamp a value within a bound.
    // The bound can be set with set_bound().
    function get_with_bound(uint256 value) public view returns (uint256) {
        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.

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

1.20.1 Function visibility

Solidity functions have a visibility specifier that restricts the scope in which they can be called. Functions can be declared public, private, internal or external with the following definitions:

- `public` functions can be called inside and outside a contract (e.g. by an RPC). They are present in the contract's ABI or IDL.
- `private` functions can only be called inside the contract they are declared.
- `internal` functions can only be called internally within the contract or by any contract inherited contract.
- `external` functions can exclusively be called by other contracts or directly by an RPC. They are also present in the contract's ABI or IDL.

Both public and external functions can be called using the syntax `this.func()`. In this case, the arguments are ABI encoded for the call, as it is treated like an external call. This is the only way to call an external function from inside the same contract it is defined. This method, however, should be avoided for public functions, as it will be more costly to call them than simply using `func()`.

If a function is defined outside a contract, it cannot have a visibility specifier (e.g. `public`).

1.20.2 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, bytes32) {
        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.

Polkadot

```
contract foo {
    function bar1(uint32 x, bool y) public returns (address, bytes32) {
        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});
    }
}
```

Solana

```
contract foo {
    function bar1(uint32 x, bool y) public returns (address, bytes32) {
        return (address(3), hex"01020304");
    }

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

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

The syntax for calling a contract is the same as that of the external call, except that it must be done on a contract type variable. Errors in external calls can be handled with *Try Catch Statement* only on Polkadot.

1.20.3 Internal calls and externals calls

An internal function call is executed by the current contract. This is much more efficient than an external call, which requires the address of the contract to call, whose arguments must be *abi encoded* (also known as serialization). Then, the runtime must set up the VM for the called contract (the callee), decode the arguments, and encode return values. Lastly, the first contract (the caller) must decode return values.

A method call done on a contract type will always be an external call. Note that `this` returns the current contract, so `this.foo()` will do an external call, which is much more expensive than `foo()`.

1.20.4 Passing accounts with external calls on Solana

The Solana runtime allows you the specify the accounts to be passed for an external call. This is specified in an array of the struct `AccountMeta`, see the section on *Builtin AccountMeta*.

```
import {AccountMeta} from 'solana';

contract SplToken {
    address constant tokenProgramId = address"TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA
↪";
    address constant SYSVAR_RENT_PUBKEY = address
↪"SysvarRent111111111111111111111111111111111";
```

(continues on next page)

(continued from previous page)

```
    struct InitializeMintInstruction {
        uint8 instruction;
        uint8 decimals;
        address mintAuthority;
        uint8 freezeAuthorityOption;
        address freezeAuthority;
    }

    function create_mint_with_freezeauthority(uint8 decimals, address mintAuthority,␣
↪address freezeAuthority) public {
        InitializeMintInstruction instr = InitializeMintInstruction({
            instruction: 0,
            decimals: decimals,
            mintAuthority: mintAuthority,
            freezeAuthorityOption: 1,
            freezeAuthority: freezeAuthority
        });

        AccountMeta[2] metas = [
            AccountMeta({pubkey: instr.mintAuthority, is_writable: true, is_signer:␣
↪false}),
            AccountMeta({pubkey: SYSVAR_RENT_PUBKEY, is_writable: false, is_signer:␣
↪false})
        ];

        tokenProgramId.call{accounts: metas}(instr);
    }
}
```

If `{accounts}` is not specified, all accounts passed to the current transaction are forwarded to the call.

1.20.5 Passing seeds with external calls on Solana

The Solana runtime allows you to specify the seeds to be passed for an external call. This is used for program derived addresses: the seeds are hashed with the calling program id to create program derived addresses. They will automatically have the signer bit set, which allows a contract to sign without using any private keys.

```
import 'solana';

contract c {
    address constant token = address"mv3ekLzLbnVPNxjSKvqBpU3ZeZXPQdEC3bp5MDEBG68";

    function test(address addr, address addr2, bytes seed) public {
        bytes instr = new bytes(1);

        instr[0] = 1;

        AccountMeta[2] metas = [
            AccountMeta({pubkey: addr, is_writable: true, is_signer: true}),
            AccountMeta({pubkey: addr2, is_writable: true, is_signer: true})
```

(continues on next page)

(continued from previous page)

```
        ];

        token.call{accounts: metas, seeds: [ [ "test", seed ], [ "foo", "bar "] ]}
↪(instr);
    }
}
```

Now if the program derived address for the running program id and the seeds match the address `addr` and `addr2`, then then the called program will run with signer and writable bits set for `addr` and `addr2`. If they do not match, the Solana runtime will detect that the `is_signer` is set without the correct signature being provided.

The seeds can provided in any other, which will be used to sign for multiple accounts. In the example above, the seed `"test"` is concatenated with the value of `seed`, and that produces one account signature. In adition, `"foo"` is concatenated with `"bar"` to produce `"foobar"` and then used to sign for another account.

The `seeds:` call parameter is a slice of bytes slices; this means the literal can contain any number of elements, including 0 elements. The values can be `bytes` or anything that can be cast to `bytes`.

1.20.6 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 {
        // ...
    }
}
```

Note: The gas cannot be set on Solana for external calls.

1.20.7 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 Polkadot, the call will be reverted.

1.20.8 Overriding function selector

When a function is called, the function selector and the arguments are serialized (also known as abi encoded) and passed to the program. The function selector is what the runtime program uses to determine what function was called. On Polkadot, the function selector is generated using a deterministic hash value of the function name and the arguments types. On Solana, the selector is known as discriminator.

The selector value can be overriden with the annotation `@selector([0xde, 0xad, 0xbe, 0xa1])`.

```
contract foo {
    // The selector attribute can be an array of values (bytes)
    @selector([1, 2, 3, 4])
    function get_foo() pure public returns (int) {
        return 102;
    }

    @selector([0x05, 0x06, 0x07, 0x08])
    function get_bar() pure public returns (int) {
        return 105;
    }
}
```

The given example only works for Polkadot, whose selectors are four bytes wide. On Solana, they are eight bytes wide.

Only `public` and `external` functions have a selector, and can have their selector overriden. On Polkadot, constructors have selectors too, so they can also have their selector overriden. If a function overrides another one in a base contract, then the selector of both must match.

Warning: On Solana, changing the selector may result in a mismatch between the contract metadata and the actual contract code, because the metadata does not explicitly store the selector.

Use this feature carefully, as it may either break a contract or cause undefined behavior.

1.20.9 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(int256 val) public returns (int256) {
```

(continues on next page)

(continued from previous page)

```
        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 = int64(abs(x));
    }
}
```

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

Both Polkadot and Solana runtime require unique function names, so overloaded function names will be mangled in the ABI or the IDL. The function name will be concatenated with all of its argument types, separated by underscores, using the following rules:

- Struct types are represented by their field types (preceded by an extra underscore).
- Enum types are represented as their underlying `uint8` type.
- Array types are recognizable by having `Array` appended.
- Fixed size arrays will additionally have their length appended as well.

The following example illustrates some overloaded functions and their mangled name:

```
enum E {
    v1,
    v2
}
struct S {
    int256 i;
    bool b;
    address a;
}

contract C {
    // foo_
    function foo() public pure {}

    // foo_uint256_addressArray2Array
    function foo(uint256 i, address[2][] memory a) public pure {}

    // foo_uint8Array2__int256_bool_address
    function foo(E[2] memory e, S memory s) public pure {}
```

(continues on next page)

(continued from previous page)

```
}
```

1.20.10 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() public only_owner {
        // ...
    }
}
```

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.

On Solana, `msg.sender` does not exist, so the usual way to implement a similar test is using an *authority* accounts rather than an owner account.

```
import 'solana';

contract AuthorityExample {
    address authority;
    uint64 counter;

    constructor(address initial_authority) {
        authority = initial_authority;
    }

    @signer(authorityAccount)
    function set_new_authority(address new_authority) external {
        assert(tx.accounts.authorityAccount.key == authority && tx.accounts.
↪authorityAccount.is_signer);
        authority = new_authority;
    }

    @signer(authorityAccount)
    function inc() external {
        assert(tx.accounts.authorityAccount.key == authority && tx.accounts.
↪authorityAccount.is_signer);
        counter += 1;
    }
```

(continues on next page)

(continued from previous page)

```
    function get() public view returns (uint64) {
        return counter;
    }
}
```

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) public check_price(price) {
        // ...
    }
}
```

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) public only_owner check_price(price) {
        // ...
    }
}
```

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

```
abstract contract base {
    address owner;
```

(continues on next page)

(continued from previous page)

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

    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) public only_owner check_price(price) {
        // ...
    }
}
```

1.20.11 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`.

Polkadot

```
contract A {
    function test(B v) public {
        // 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);
```

(continues on next page)

(continued from previous page)

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

        assert(res == 8);
    }
}

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

Solana

```
contract A {
    function test(address v) public {
        // 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(
            "global:foo",
            uint32(3),
            uint32(5)
        );

        (bool success, bytes rawresult) = v.call{accounts: []}(data);

        assert(success == true);

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

        assert(res == 8);
    }
}

contract B {
    function foo(uint32 a, uint32 b) pure 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);
}
```

External calls with the `call()` method on Solana must have the `accounts` call argument, regardless of the callee function visibility, because the compiler has no information about the caller function to generate the `AccountMeta` array automatically.

```
function test(address foo, bytes rawcalldata) public {
    (bool success, bytes rawresult) = foo.call{accounts: []}(rawcalldata);
}
```

1.20.12 Calling an external function using `delegatecall`

External functions can also be called using `delegatecall`. The difference to a regular `call` is that `delegatecall` executes the callee code in the context of the caller:

- The callee will read from and write to the *caller* storage.
- `value` can't be specified for `delegatecall`; instead it will always stay the same in the callee.
- `msg.sender` does not change; it stays the same as in the callee.

Refer to the contracts pallet and Ethereum Solidity documentations for more information.

`delegatecall` is commonly used to implement re-usable libraries and upgradeable contracts.

```
function delegate(
    address callee,
    bytes input
) public returns(bytes result) {
    (bool ok, result) = callee.delegatecall(input);
    require(ok);
}
```

Note: `delegatecall` is not available on Solana.

Note: On Polkadot, specifying `gas` won't have any effect on `delegatecall`.

1.20.13 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(int32 x) public {
        bar = x;
    }

```

(continues on next page)

(continued from previous page)

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

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

Note: On Solana, there is no mechanism to have some code executed if an account gets credited. So, *receive()* functions are not supported.

1.21 Managing values

1.21.1 Sending and receiving value

Value in Solidity is represented by `uint128`.

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

1.21.2 Checking your balance

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

Note: Polkadot 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.

Note: On Solana, checking the balance of an account different than the program account requires that it be passed as an AccountMeta during the transaction. It is not common practice for the program account to hold native Solana tokens.

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

1.21.3 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*.

1.21.4 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*.

1.21.5 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.

Note: On Solana, `send()` and `transfer()` can only transfer native tokens between accounts owned by the contract's program account, since only the account owner can modify its balance. Use the *system instruction library* to transfer

native tokens between accounts owned by Solana's system program.

1.22 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.

1.22.1 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.

Note: `gasleft()` is not available on Solana.

Gasprice is not used on Solana. There is compute budget which may not be exceeded, but there is no charge based on compute units used.

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 Solana. There is the recent block hashes account that looks useful at first glance, however it is not usable because:

- This account is deprecated.
- It does not give any slot of block number, so it is not possible to provide a matching function signature.

`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 (Polkadot) or bytes8 (Solana) `msg.sig`
Function selector (or discriminator for Solana) from the encoded calldata, e.g. the first four or eight bytes. This might be 0 if no function selector was present. In Ethereum, constructor calls do not have function selectors but in Polkadot they do. On Solana, selectors are called discriminators.

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 Polkadot, see the warning box below.

Note: `tx.gasprice` is not available on Solana.

gasprice is not used on Solana. There is compute budget which may not be exceeded, but there is no charge based on compute units used.

---

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

Warning: On Polkadot, 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 Polkadot or Solana.

AccountInfo[] `tx.accounts`
Only available on Solana. See *Builtin AccountInfo*. Here is an example:

```
import {AccountInfo} from "solana";

contract SplToken {
    function get_token_account(address token)
        internal
        view
        returns (AccountInfo)
    {
        for (uint64 i = 0; i < tx.accounts.length; i++) {
            AccountInfo ai = tx.accounts[i];
            if (ai.key == token) {
                return ai;
            }
        }
```

(continues on next page)

(continued from previous page)

```
        revert("token not found");
    }

    function total_supply(address token) public view returns (uint64) {
        AccountInfo account = get_token_account(token);

        return account.data.readUint64LE(33);
    }
}
```

`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.

Note: Solana requires the clock account to present in the account for the instruction to use any of the `block` fields.

On Solana, `block.number` gives the slot number rather than the block height. For processing, you want to use the slot rather the block height. Slots include empty blocks, which do not count towards the block height.

Solana

uint64 `block.slot`
The current slot. This is an alias for `block.number`.

Polkadot

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.

1.22.2 Error handling

assert(bool)

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

```
abstract contract c {
    constructor(int256 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.

```
abstract contract x {
    constructor(address foobar) {
        if (foobar == 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.

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

1.22.3 ABI encoding and decoding

The ABI encoding depends on the target being compiled for. Polkadot uses the SCALE Codec.

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 Polkadot, foo will be `hex"f100"`. On Ethereum this will be `hex"0000000000000000000000000000000000000000000000000000000000000000`

abi.encodeWithSelector(selector, ...)

ABI encodes the arguments with the function selector, which is known as the discriminator on Solana. After the selector, any number of arguments can be provided.

```
// An eight-byte selector (discriminator) is exclusive for Solana.
// On Polkadot, the selector contains four bytes. hex"01020304" is an example.
bytes foo = abi.encodeWithSelector(hex"0102030405060708", uint16(0xff00));
```

On Solana, foo will be `hex"0807060504030201 00ff"`. In addition, a discriminator for a Solidity function on Solana are the first eight bytes of the sha-256 hash of its name converted to camel case and preceded by the prefix `global:`, as the following:

```
bytes8 discriminator = bytes8(sha256(bytes("global:myFunctionName")));
```

abi.encodeWithSignature(string signature, ...)

ABI encodes the arguments with the hash of the signature. After the signature, any number of arguments can be provided.

On Polkadot, the signature is the name of the function followed by its arguments, for example:

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

`foo` will be `hex"e934aa71_0101_0000__0000_0000"`. This is equivalent to `abi.encodeWithSelector(bytes4(keccak256("test2(uint64)")), ...)`.

On Solana, the signature is known as the discriminator image. It is the function name without any arguments, converted to camel case, and preceded by the prefix `global:`. For example, if you had the function `foo_bar(uint64)`, the discriminator image would be `global:fooBar`.

```
bytes foo = abi.encodeWithSignature("global:fooBar", uint64(257));
```

This builtin is equivalent to `abi.encodeWithSelector(bytes8(sha256(bytes("global:fooBar"))), ...)` for Solana.

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.encodePacked(uint16(0xff00), "ABCD");
```

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

abi.encodeCall(function, ...)

ABI encodes the function call to the function which should be specified as `ContractName.FunctionName`. The arguments are cast and checked against the function specified as the first argument.

```
contract c {
    function f1() public {
        bytes foo = abi.encodeCall(c.bar, 102, true);
    }

    function bar(int256 a, bool b) public {}
}
```

Hash

Only available on Polkadot, it represents the `Hash` type from `ink_primitives` via user type definition. Its underlying type is `bytes32`, but it will be reported correctly as the `Hash` type in the metadata.

```
import "polkadot";

contract c {
    bytes32 current;

    function set(Hash h) public returns (Hash) {
        current = Hash.unwrap(h);
        return Hash.wrap(current);
    }
}
```

chain_extension(uint32 ID, bytes input) returns (uint32, bytes)

Only available on Polkadot. Call the chain extension with the given `ID` and `input` data. Returns the return value from the chain extension and the output data.

This function is a low level interface. The caller is responsible for encoding the input and decoding the output correctly. We expect parachain authors to write their own higher level libraries on top.

Warning: This function calls the runtime API call_chain_extension.

It assumes that the implementation of the chain extension

- reads the input from the `input_ptr` parameter, used as a buffer pointer
- writes potential output into the buffer found at the `output_ptr` pointer
- respects the output buffer length in `output_len_ptr` to prevent OOB writes. The output buffer is 16KB in size.
- writes the amount of bytes written to `output_ptr` into the buffer at `output_len_ptr`

Unlike with other runtime API calls, the contracts pallet can not guarantee this behaviour. Instead, it's specific to the targeted chain runtime. Hence, when using this builtin, you must be sure that the implementation being called underneath is compatible.

The following example demonstrates the usage of this builtin function. It shows how the chain extension example from the ink! documentation looks like in a solidity contract:

```
import "polkadot";

contract Foo {
    // Call the "rand-extension" example chain extension demonstrated here:
    // https://use.ink/macros-attributes/chain-extension
    //
    // This chain extension is registered under ID 1101.
    // It takes a bytes32 as input seed and returns a pseudo random bytes32.
    function fetch_random(bytes32 _seed) public returns (bytes32) {
        bytes input = abi.encode(_seed);
        (uint32 ret, bytes output) = chain_extension(1101, input);

        assert(ret == 0); // The fetch-random chain extension always returns 0
        bytes32 random = abi.decode(output, (bytes32));

        print("psuedo random bytes: {}".format(random));
        return random;
    }
}
```

is_contract(address AccountId) returns (bool)

Only available on Polkadot. Checks whether the given address is a contract address.

set_code_hash(uint8[32] hash) returns (uint32)

Only available on Polkadot. Replace the contract's code with the code corresponding to `hash`. Assumes that the new code was already uploaded, otherwise the operation fails. A return value of 0 indicates success; a return value of 7 indicates that there was no corresponding code found.

Note: This is a low level function. We strongly advise consulting the underlying API documentation to obtain a full understanding of its implications.

This functionality is intended to be used for implementing upgradeable contracts. Pitfalls generally applying to writing upgradeable contracts must be considered whenever using this builtin function, most notably:

- The contract must safeguard access to this functionality, so that it is only callable by priviledged users.
- The code you are upgrading to must be storage compatible with the existing code.
- Constructors and any other initializers, including initial storage value definitions, won't be executed.

1.22.4 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 Polkadot.

---

blake2_256(bytes)

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

Note: This function is only available on Polkadot.

signatureVerify(address public_key, bytes message, bytes signature)

Verify the ed25519 signature given the public key, message, and signature. This function returns `true` if the signature matches, `false` otherwise.

The transactions which executes this function, needs an ed25519 program instruction with matching public key, message, and signature. In order to examine the instruction, the instructions sysvar needs be in the accounts for the Solidity instruction as well.

Note: This function is only available on Solana.

1.22.5 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.

1.22.6 Encoding and decoding values from bytes buffer

The `abi.encode()` and friends functions do not allow you to write or read data from an arbitrary offset, so the Solang dialect has the following functions. These methods are available on a `bytes` type.

These functions are inspired by the node buffer api.

```
contract c {
    function f() public returns (bytes) {
        bytes data = new bytes(10);
        data.writeUint32LE(102, 0);
        data.writeUint64LE(0xdeadcafe, 4);
        return data;
    }

    function g(bytes data) public returns (uint64) {
        return data.readUint64LE(1);
    }
}
```

readInt8(uint32 offset)

Read a signed `int8` from the specified offset.

readInt16LE(uint32 offset)

Read a signed `int16` from the specified offset in little endian order.

readInt32LE(uint32 offset)

Read a signed `int32` from the specified offset in little endian order.

readInt64LE(uint32 offset)

Read a signed `int64` from the specified offset in little endian order.

readInt128LE(uint32 offset)

Read a signed `int128` from the specified offset in little endian order.

readInt256LE(uint32 offset)

Read a signed `int256` from the specified offset in little endian order.

readUint16LE(uint32 offset)

Read an unsigned `uint16` from the specified offset in little endian order.

readUint32LE(uint32 offset)

Read an unsigned `uint32` from the specified offset in little endian order.

readUint64LE(uint32 offset)

Read an unsigned `uint64` from the specified offset in little endian order.

readUint128LE(uint32 offset)

Read an unsigned `uint128` from the specified offset in little endian order.

readUint256LE(uint32 offset)

Read an unsigned `uint256` from the specified offset in little endian order.

readAddress(uint32 offset)

Read an `address` from the specified offset.

writeInt8(int8 value, uint32 offset)

Write a signed `int8` to the specified offset.

writeInt16LE(int16 value, uint32 offset)

Write a signed `int16` to the specified offset in little endian order.

writeInt32LE(int32 value, uint32 offset)

Write a signed `int32` to the specified offset in little endian order.

writeInt64LE(int64 value, uint32 offset)

Write a signed `int64` to the specified offset in little endian order.

writeInt128LE(int128 value, uint32 offset)

Write a signed `int128` to the specified offset in little endian order.

writeInt256LE(int256 value, uint32 offset)

Write a signed `int256` to the specified offset in little endian order.

writeUint16LE(uint16 value, uint32 offset)

Write an unsigned `uint16` to the specified offset in little endian order.

writeUint32LE(uint32 value, uint32 offset)

Write an unsigned `uint32` to the specified offset in little endian order.

writeUint64LE(uint64 value, uint32 offset)

Write an unsigned `uint64` to the specified offset in little endian order.

writeUint128LE(uint128 value, uint32 offset)

Write an unsigned `uint128` to the specified offset in little endian order.

writeUint256LE(uint256 value, uint32 offset)

Write an unsigned `uint256` to the specified offset in little endian order.

writeAddress(address value, uint32 offset)

Write an `address` to the specified offset.

writeString(string value, uint32 offset)

Write the characters of a `string` to the specified offset. This function does not write the length of the string to the buffer.

writeBytes(bytes value, uint32 offset)

Write the bytes of a Solidity dynamic bytes type `bytes` to the specified offset. This function does not write the length of the byte array to the buffer.

1.22.7 Miscellaneous

print(string)

print() takes a string argument.

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

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

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

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.

Note: This function does not exist on Solana.

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.

1.23 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.
```

(continues on next page)

(continued from previous page)

```
 * @author Sean Young <sean@mess.org>
 */
contract c {
    /// @param name The name which will be greeted
    function say_hello(string name) public {
        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.

1.24 Inline Assembly

In Solidity functions, developers are allowed to write assembly blocks containing Yul code. For more information about the Yul programming language, please refer to the *yul section*.

In an assembly block, you can access solidity local variables freely and modify them as well. Bear in mind, however, that reference types like strings, vectors and structs are memory addresses in yul, so manipulating them can be unsafe unless done correctly. Any assignment to those variables will change the address the reference points to and may cause the program to crash if not managed correctly.

```
contract foo {
    struct test_stru {
        uint256 a;
        uint256 b;
    }

    function bar(uint64 a) public pure returns (uint64 ret) {
        uint64 b = 6;
        uint64[] memory vec;
        vec.push(4);
        string str = "cafe";
        test_stru tts = test_stru({a: 1, b: 2});
        assembly {
            // The following statements modify variables directly
```

(continues on next page)

(continued from previous page)

```
            a := add(a, 3)
            b := mul(b, 2)
            ret := sub(a, b)

            // The following modify the reference address
            str := 5
            vec := 6
            tts := 7
        }

        // Any access to 'str', 'vec' or 'tts' here may crash the program.
    }
}
```

Storage variables cannot be accessed nor assigned directly. You must use the `.slot` and `.offset` suffix to use storage variables. Storage variables should be read with the `sload` and saved with `sstore` builtins, but they are not implemented yet. Solang does not implement offsets for storage variables, so the `.offset` suffix will always return zero. Assignments to the offset are only allowed to Solidity local variables that are a reference to the storage.

```
contract foo {
    struct test_stru {
        uint256 a;
        uint256 b;
    }

    test_stru storage_struct;

    function bar() public view {
        test_stru storage tts = storage_struct;
        assembly {
            // The variables 'a' and 'b' contain zero
            let a := storage_struct.offset
            let b := tts.offset

            // This changes the reference slot of 'tts'
            tts.slot := 5
        }
    }
}
```

Dynamic calldata arrays should be accessed with the `.offset` and `.length` suffixes. The offset suffix returns the array's memory address. Assignments to `.length` are not yet implemented.

```
contract foo {
    struct test_stru {
        uint256 a;
        uint256 b;
    }

    test_stru storage_struct;

    function bar(int256[] calldata vl) public view {
```

(continues on next page)

(continued from previous page)

```
        test_stru storage tts = storage_struct;
        assembly {
            // 'a' contains vl memory address
            let a := vl.offset

            // 'b' contains vl length
            let b := vl.length

            // This will change the reference of vl
            vl.offset := 5
        }
        // Any usage of vl here may crash the program
    }
}
```

External functions in Yul can be accessed and modified with the `.selector` and `.address` suffixes. The assignment to those values, however, are not yet implemented.

```
contract foo {
    function sum(uint64 a, uint64 b) public pure returns (uint64) {
        return a + b;
    }

    function bar() public view {
        function (uint64, uint64) external returns (uint64) fPtr = this.sum;
        assembly {
            // 'a' contains 'sum' selector
            let a := fPtr.selector

            // 'b' contains 'sum' address
            let b := fPtr.address
        }
    }
}
```

1.25 Overview of Yul

Yul, also know as EVM assembly, is a low level language for creating smart contracts and for providing more control over the execution environment when using Solidity as the primary language. Although it enables more possibilities to manage memory, using Yul does not imply a performance improvement. In Solang, all Yul constructs are processed using the same pipeline as Solidity.

The support for Yul is only partial. We support all statements, except for the switch-block. In addition, some Yul builtins are not yet implemented. In the following sections, we'll describe the state of the compatibility of Yul in Solang.

1.26 Statements

1.26.1 For-loop

For-loops are completely supported in Solang. `continue` and `break` statements are also supported. The syntax for `for` is quite different from other commonly known programming languages. After the `for` keyword, the lexer expects a yul block delimited with curly brackets that initializes variables for the loop. This block can have as many statements as needed and is always executed.

After the initialization block, there should be an Yul expression that contains the loop stopping condition. Then comes the update block, which contains all the statements executed after the main body block, but before the condition check. The body block is a set of instructions executed during each iteration.

```
{
    function foo()
    {
        // Simple for loop
       for {let i := 0} lt(i, 10) {i := add(i, 10)} {
            let p := funcCall(i, 10)
            if eq(p, 5) {
                continue
            }

            if eq(p, 90) {
                break
            }
        }

        let a := 0
        // More complex loops are also possible
        for {
            let i := 0
            let j := 3
            i := add(j, i)
        } or(lt(i, 10), lt(j, 5)) {
            i := add(i, 1)
            j := add(j, 3)
        } {
            a := add(a, mul(i, j))
        }
    }
}
```

1.26.2 If-block

If-block conditions in Yul cannot have an *else*. They act only as a branch if the condition is true and are totally supported in Solang.

```
{
    if eq(5, 4) {
        funcCall(4, 3)
    } // There cannot be an 'else' here
}
```

1.26.3 Switch

Switch statements are not yet supported in Solang. If there is urgent need to support them, please, file a GitHub issue in the repository.

1.26.4 Blocks

There can be blocks of code within Yul, defined by curly brackets. They have their own scope and any variable declared inside a block cannot be accessed outside it. Statements inside a block can access outside variables, though.

```
{
    function foo() -> ret {
        let g := 0
        { // This is a code block
            let r := 7
            ret := mul(g, r)
        }
    }
}
```

1.26.5 Variable declaration

Variables can be declared in Yul using the *let* keyword. Multiple variables can be declared at the same line if there is no initializer or the initializer is a function that returns multiple values.

The default type for variables in Yul is `u256`. If you want to declare a variable with another type, use the colon. Note that if the variable type and the type of the right hand side of the assignment do not match, there will be an implicit type conversion to the correct type.

```
{
    let a, b, c
    let d := funCall()
    let e : u64 := funcCall()
    let g, h, i := multipleReturns()
    let j : u32, k : u8 := manyReturns()
}
```

1.26.6 Assignments

Variables can be assignment using the `:=` operator. If the types do not match, the compiler performs an implicit conversion, so that the right hand side type matches that of the variable. Multiple variables can be assigned in a single line if the right hand side is a function call that returns multiple values.

```
{
    a := 6
    c, d := multipleReturns()
}
```

1.26.7 Function calls

Function calls in Yul are identified by the use of parenthesis after an identifier. Standalone function calls must not return anything. Functions that have multiple returns can only appear in an assignment or definition of multiple variables.

```
{
    noReturns()
    a := singleReturn()
    // multipleReturns() cannot be inside 'add'
    let g := add(a, singleReturn())
    f, d, e := multipleReturns()
}
```

1.27 Types

The default type for variables in Yul is `u256`. If there is no type specified, the compiler will default to that integer type. Variables can have a type specified during their declaration using the following syntax:

```
{
    let a : u32, b : s64, d : u128, e : bool := multipleReturns()
}
```

Yul allows signed and unsigned integer types, in addition to the boolean type. If a conversion from an integer to a boolean is necessary, the compiler does the following operation `number != 0`.

The unsigned types are the following: `u8`, `u32`, `u64`, `u128` and `u256`.

The signed types are the following: `s8`, `s32`, `s64`, `s128`, `s256`.

1.28 Functions

Functions in Yul cannot access any variable outside their scope, i.e. they can only operate with the variables they receive as arguments. You can define types for arguments and returns. If no type is specified, the compiler will default to `u256`.

Warning: Yul functions are only available within the scope there are defined. They cannot be accessed from Solidity and from other inline assembly blocks, even if they are contained within the same Solidity function.

Differently from solc, Solang allows name shadowing inside Yul functions. As they cannot access variables declared outside them, a redefinition of an outside name is allowed, if it has not been declared within the function yet. Builtin function names cannot be overdriven and verbatim functions supported by Solc are not implemented in Solang. `verbatim`, nevertheless, is still a reserved keyword and cannot be the prefix of variable or function names.

Function calls are identified by a name followed by parenthesis. If the types of the arguments passed to function calls do not match the respective parameter's type, Solang will implicitly convert them. Likewise, the returned values of function calls will be implicitly converted to match the type needed in an expression context.

```
{
    // return type defaulted to u256
    function noArgs() -> ret
    {
        ret := 2
    }

    // Parameters defaulted to u256 and ret has type u64
    function sum(a, b) -> ret : u64
    {
        ret := add(b, a)
    }

    function getMod(c : s32, d : u128, e) -> ret1, ret2 : u64
    {
        ret1 := mulmod(c, d, e)
        ret2 := addmod(c, d, e)
    }

    function noReturns(a, b)
    {
        // Syntax of function calls
        let x := noArgs()
        // Arguments will be implicitly converted form u256 to s32 and u128,␣
↪respectively.
        // The returns will also be converted to u256
        let c, d := getMod(a, b)
        {
            // 'doThis' cannot be called from outside the block defined the by␣
↪curly brackets.
            function doThis(f, g) -> ret {
                ret := sdiv(f, g)
            }
        }
        // 'doThat' cannot be called from outside 'noReturns'
        function doThat(f, g) -> ret {
            ret := smod(g, f)
        }
    }
}
```

1.29 Builtins

Most operations in Yul are performed via builtin functions. Solang supports most builtins, however memory operations and chain operations are not implemented. Yul builtins are low level instructions and many are ethereum specific. On Solana and Polkadot, some builtins, like `delegatecall` and `staticcall`, for instance, are not available because the concept they implement does not exist in neither chains.

Warning: In addition to nonexistent builtins, due to low-level differences between blockchain virtual machines, it is impossible to replicate the builtin's behavior outside Ethereum. `pop`, for example, removes an item from the stack in EVM, however, in Solana there is no stack, for its virtual machine is register based.

This is the comprehensive list of the existing Yul builtins and their compatibility on Solang. Arithmetic operations always return the widest integer between the arguments. Signed numbers are represented in two's complement. The descriptions in the table have been slightly modified from the Solc documentation.

Builtin	Returns	Explanation	Availability
stop()	None	stop execution	No
add(x, y)	Integer	x + y	Yes
sub(x, y)	Integer	x - y	Yes
mul(x, y)	Integer	x * y	Yes
div(x, y)	Integer	x / y or 0 if y == 0	Yes
sdiv(x, y)	Integer	x / y or 0 if y == 0, for signed integers	Yes
mod(x, y)	Integer	x % y or 0 if y == 0	Yes
smod(x, y)	Integer	x % y or 0 if y == 0, for signed integers	Yes
exp(x, y)	Integer	x to the power of y	Yes
not(x)	Integer	negation of every bit of x	Yes
lt(x, y)	Bool	x < y	Yes
gt(x, y)	Bool	x > y	Yes
slt(x, y)	Bool	x < y, for signed integers	Yes
sgt(x, y)	Bool	x > y, for signed integers	Yes
eq(x, y)	Bool	x == y	Yes
iszero(x)	Bool	x == 0	Yes
and(x, y)	Integer	bitwise AND of x and y	Yes
or(x, y)	Integer	bitwise OR of x and y	Yes
xor(x, y)	Integer	bitwise XOR of x and y	Yes
byte(n, x)	Integer	nth byte of x, where 0 is the most significant bit	Yes
shl(x, y)	Integer	y << x	Yes
shr(x, y)	Integer	y >> x	Yes
sar(x, y)	Integer	y >> x, for signed integers	Yes
addmod(x, y, m)	Integer	(x + y) % m or 0 if m == 0	Yes
mulmod(x, y, m)	Integer	(x * y) % m or 0 if m == 0	Yes
signextend(i, x)	Integer	sign extend from (i*8+7)th bit, where 0th is the least significant bit	No

continues on next page

Table 1 – continued from previous page

Builtin	Returns	Explanation	Availability
keccak256(p, n)	Integer	keccak(mem[p…(p+n)))	No
pc()	Integer	program counter	No
pop(x)	None	discard value x from the stack	No
mload(p)	Integer	load from memory mem[p…(p+32))	No
mstore(p, v)	None	store v in memory mem[p…(p+32))	No
mstore8(p, v)	None	store v & 0xff byte in memory mem[p]	No
sload(p)	Integer	Load from storage slot p	No
sstore(p, v)	Integer	store v in storage slot p	No
msize()	Integer	largest accessed memory index	No
gas()	Integer	gas still available to execution	Yes
address()	Integer	address of the current contract	Yes
balance(a)	Integer	balance at address a	Yes
selfbalance()	Integer	equivalent to balance(address())	Yes
caller()	Integer	call sender (excluding `delegatecall`)	Yes
callvalue()	Integer	wei sent together with the current call	Yes
calldataload(p)	Integer	load call data starting from position p	No
calldatasize()	Integer	size of call data in bytes	No
calldatacopy(t, f, s)	None	copy s bytes from calldata at position f to mem at position t	No
codesize()	Integer	size of the code of the current contract or execution context	No
codecopy(t, f, s)	None	copy s bytes from code at position f to mem at position t	No
extcodesize(a)	Integer	size of the code at address a	No
extcodecopy(a, t, f, s)	None	like codecopy(t, f, s), but take code at address a	No

continues on next page

Table 1 – continued from previous page

Builtin	Returns	Explanation	Availability
returndatasize()	Integer	size of the last returndata	No
returndatacopy(t, f, s)	None	copy s bytes from returndata at position f to mem at position t	No
extcodehash(a)	Integer	code hash of address a	No
create(v, p, n)	Integer	create new contract with code mem[p…(p+n)) and send v wei and return the new address; returns 0 on error	No
create2(v, p, n, s)	Integer	create new contract with code mem[p…(p+n)) at address resulting from the keccak256 hash of 0xff.this.s.keccak256(mem[and send v wei and return the new address, where `0xff` is a 1 byte value, `this` is the current contract's address as a 20 byte value and `s` is a big-endian 256-bit value; returns 0 on error	No

continues on next page

Table 1 – continued from previous page

Builtin	Returns	Explanation	Availability
call(g, a, v, in, insize, out, outsize)	Integer	call contract at address a with in mem[in…(in+insize)) providing g gas and v wei and output area mem[out…(out+outsize)) returning 0 on error (eg. out of gas) and 1 on success	No
callcode(g, a, v, in, insize, out, outsize)	Integer	identical to `call` but only use the code from a and stay in the context of the current contract otherwise	No
delegatecall(g, a, in, insize, out, outsize)	Integer	identical to `callcode` but also keep `caller` and `callvalue`	No
staticcall(g, a, in, insize, out, outsize)	Integer	identical to `call` but do not allow state modifications	No
return(p, s)	None	end execution, return data mem[p…(p+s))	No
revert(p, s)	None	end execution, revert state changes, return data mem[p…(p+s))	No
selfdestruct(a)	None	end execution, destroy current contract and send funds to a	No
invalid()	None	end execution with invalid instruction	Yes
log0(p, s)	None	log without topics and data mem[p…(p+s)]	No

continues on next page

Table 1 – continued from previous page

Builtin	Returns	Explanation	Availability
log1(p, s, t1)	None	log with topic t1 and data mem[p…(p+s)]	No
log2(p, s, t1, t2)	None	log with topics t1, t2 and data mem[p…(p+s))	No
log3(p, s, t1, t2, t3)	None	log with topics t1, t2, t3 and data mem[p…(p+s))	No
log4(p, s, t1, t2, t3, t4)	None	log with topics t1, t2, t3, t4 and data mem[p…(p+s))	No
chainid()	Integer	ID of the executing chain	No
basefee()	Integer	current block's base fee	No
origin()	Integer	transaction sender	Yes
gasprice()	Integer	gas price of the transaction	Yes
blockhash(b)	Integer	hash of block nr b - only for last 256 blocks excluding current	Yes
coinbase()	Integer	current mining beneficiary	Yes
timestamp()	Integer	timestamp of the current block in seconds since the epoch	Yes
number()	Integer	current block number	Yes
difficulty()	Integer	difficulty of the current block	Yes
gaslimit()	Integer	block gas limit of the current block	Yes

1.30 Code Generation Options

There are compiler flags to control code generation. They can be divided into two categories:

- Optimizer passes are enabled by default and make the generated code more optimal.
- Debugging options are enabled by default. They can be disabled in release builds, using *–release* CLI option, to decrease gas or compute units usage and code size.

1.30.1 Optimizer Passes

Solang generates its own internal IR, before the LLVM IR is generated. This internal IR allows us to do several optimizations which LLVM cannot do, since it is not aware of higher-level language constructs.

Arithmetic of large integers (larger than 64 bit) has special handling, since LLVM cannot generate them. So we need to do our own optimizations for these types, and we cannot rely on LLVM.

Constant Folding Pass

There is a constant folding (also called constant propagation) pass done, before all the other passes. This helps arithmetic of large types, and also means that the functions are constant folded when their arguments are constant. For example:

```
bytes32 hash = keccak256('foobar');
```

This is evaluated at compile time. You can see this in the Visual Studio Code extension by hover over *hash*; the hover will tell you the value of the hash.

Strength Reduction Pass

Strength reduction is when expensive arithmetic is replaced with cheaper ones. So far, the following types of arithmetic may be replaced:

- 256 or 128 bit multiply maybe replaced by 64 bit multiply or shift
- 256 or 128 bit divide maybe replaced by 64 bit divide or shift
- 256 or 128 bit modulo maybe replaced by 64 bit modulo or bitwise and

```
contract test {
    function f() public {
        for (uint256 i = 0; i < 10; i++) {
            // this multiply can be done with a 64 bit instruction
            g(i * 100);
        }
    }

    function g(uint256 v) internal {
        // ...
    }
}
```

Solang uses reaching definitions to track the known bits of the variables; here solang knows that i can have the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and the other operand is always 100. So, the multiplication can be done using a single 64 bit multiply instruction. If you hover over the * in the Visual Studio Code you will see this noted.

Dead Storage pass

Loading from contract storage, or storing to contract storage is expensive. This optimization removes any redundant load from and store to contract storage. If the same variable is read twice, then the value from the first load is re-used. Similarly, if there is are two successive stores to the same variable, the first one is removed as it is redundant. For example:

```
contract test {
    int256 a;

    // this function reads a twice; this can be reduced to one load
    function redundant_load() public returns (int256) {
        return a + a;
    }

    // this function writes to contract storage thrice. This can be reduced to one
    function redundant_store() public {
        delete a;
        a = 1;
        a = 2;
    }
}
```

This optimization pass can be disabled by running *solang –no-dead-storage*. You can see the difference between having this optimization pass on by comparing the output of *solang –no-dead-storage –emit cfg foo.sol* with *solang –emit cfg foo.sol*.

Vector to Slice Pass

A *bytes* or *string* variable can be stored in a vector, which is a modifyable in-memory buffer, and a slice which is a pointer to readonly memory and an a length. Since a vector is modifyable, each instance requires a allocation. For example:

```
contract test {
    function can_be_slice() public {
        // v can just be a pointer to constant memory and an a length indicator
        string v = "Hello, World!";

        print(v);
    }

    function must_be_vector() public {
        // if v is a vector, then it needs to allocated and default value copied.
        string v = "Hello, World!";

        // bs is copied by reference is now modifyable
        bytes bs = bytes(v);

        bs[1] = 97;

        print(v);
    }
}
```

This optimization pass can be disabled by running *solang –no-vector-to-slice*. You can see the difference between having this optimization pass on by comparing the output of *solang –no-vector-to-slice –emit cfg foo.sol* with *solang –emit cfg foo.sol*.

Unused Variable Elimination

During the semantic analysis, Solang detects unused variables and raises warnings for them. During codegen, we remove all assignments that have been made to this unused variable. There is an example below:

```
contract test {
    function test1(int256 a) public pure returns (int256) {
        int256 x = 5;
        x++;
        if (a > 0) {
            x = 5;
        }

        a = (x = 3) + a * 4;

        return a;
    }
}
```

The variable 'x' will be removed from the function, as it has never been used. The removal won't affect any expressions inside the function.

Common Subexpression Elimination

Solang performs common subexpression elimination by doing two passes over the CFG (Control Flow Graph). During the first one, it builds a graph to track existing expressions and detect repeated ones. During the second pass, it replaces the repeated expressions by a temporary variable, which assumes the value of the expression. To disable this feature, use *solang –no-cse*.

Check out the example below. It contains multiple common subexpressions:

```
contract test {
    function csePass(int256 a, int256 b) public pure returns (int256) {
        int256 x = a * b - 5;
        if (x > 0) {
            x = a * b - 19;
        } else {
            x = a * b * a;
        }

        return x + a * b;
    }
}
```

The expression *a*b* is repeated throughout the function and will be saved to a temporary variable. This temporary will be placed wherever there is an expression *a*b*. You can see the pass in action when you compile this contract and check the CFG, using *solang –emit cfg*.

Array Bound checks optimization

Whenever an array access is done, there must be a check for ensuring we are not accessing beyond the end of an array. Sometimes, the array length could be known. For example:

```
contract c {
    function test() public returns (int256[]) {
        int256[] array = new int256[](3);
        array[1] = 1;
        return array;
    }
}
```

In this example we access `array` element 1, while the array length is 3. So, no bounds checks are necessary and the code will more efficient if we do not emit the bounds check in the compiled contract.

The array length is tracked in an invisible temporary variable, which is always kept up to date when, for example, a `.pop()` or `.push()` happens on the array or an assignment happens. Then, when the bounds check happens, rather than retrieving the array length from the array at runtime, bounds check becomes the constant expression *1 < 3* which is always true, so the check is omitted.

This also means that, whenever the length of an array is accessed using '.length', it is replaced with a constant.

Note that this optimization does not cover every case. When an array is passed as a function argument, for instance, the length is unknown.

1.30.2 `wasm-opt` optimization passes

For the Polkadot target, optimization passes from the binaryen `wasm-opt` tool can be applied. This may shrink the Wasm code size and makes it more efficient.

Use the `--wasm-opt` compile flag to enable `wasm-opt` optimizations. Possible values are `0` - `4`, `s` and `z`, corresponding to the `wasm-opt` flags `-O0` - `-O4`, `-Os` and `-Oz` respectively. To learn more about the optimization levels please consult `wasm-opt --help`.

Note: In `--release` mode, if `--wasm-opt` is not specified, the level `z` ("super-focusing on code size") will be used.

1.30.3 Debugging Options

It is desirable to have access to debug information regarding the contract execution in the testing phase. Therefore, by default, debugging options are enabled; however, they can be deactivated by utilizing the command-line interface (CLI) flags. Debugging options should be disabled in release builds, as debug builds greatly increase contract size and gas consumption. Solang provides three debugging options, namely debug prints, logging API return codes, and logging runtime errors. For more flexible debugging, Solang supports disabling each debugging feature on its own, as well as disabling them all at once with the `--release` flag.

Print Function

Solang provides a *print(string)* which is enabled by default. The `no-print` flag will instruct the compiler not to log debugging prints in the environment.

Log runtime API call results

Runtime API calls are not guaranteed to succeed. By design, the low level results of these calls are abstracted away in Solidity. For development purposes, it can be desirable to observe the low level return code of such calls. The contract will print the return code of runtime calls by default, and this feature can be disabled by providing the `--no-log-api-return-codes` flag.

Note: This is only implemented for the Polkadot target.

Log Runtime Errors

In most cases, contract execution will emit a human readable error message in case a runtime error is encountered. The error is printed out alongside with the filename and line number that caused the error. This feature is enabled by default, and can be disabled by the `--no-log-runtime-errors` flag.

Release builds:

Release builds must not contain any debugging related logic. The `--release` flag will turn off all debugging features, thereby reducing the required gas and storage.

1.31 Solang Test Suite

Solang has a few test suites. These are all run on each pull request via github actions.

1.31.1 Solidity parser and semantics tests

In the tests directory, there are a lot of tests which call *fn parse_and_resolve()*. This function parses Solidity, and returns the *namespace*: all the resolved contracts, types, functions, etc (as much as could be resolved), and all the compiler diagnositics, i.e. compiler warnings and errors. These tests check that the compiler parser and semantic analysis work correctly.

Note that Solidity can import other soldity files using the `import` statement. There are further tests which create a file cache with filenames and and their contents, to ensure that imports work as expected.

1.31.2 Codegen tests

The stage after semantic analysis is codegen. Codegen generates an IR which is a CFG, so it is simply called CFG. The codegen tests ensure that the CFG matches what should be created. These tests are inspired by LLVM lit tests. The tests can found in codegen_testcases.

These tests do the following:

- Look for a comment `// RUN:` and then run the compiler with the given arguments and the filename itself
- After that the output is compared against special comments:
- `// CHECK:` means that following output must be present
- `// BEGIN-CHECK:` means check for the following output but scan the output from the beginning
- `// FAIL:` will check that the command will fail (non-zero exit code) with the following output

1.31.3 Mock contract virtual machine

For Polkadot and Solana there is a mock virtual machine. System and runtime call implementations should semantically represent the real on-chain virtual machine as exact as possible. Aspects that don't matter in the context of unit testing (e.g. gas-metering) may be ignored in the mock virtual machine. For Polkadot, this uses the wasmi crate and for Solana it uses the Solana RBPF crate.

These tests consist of calling a function call *fn build_solidity()* which compiles the given solidity source code and then returns a *VM*. This *VM* can be used to deploy one of the contract, and test various functions like contract storage, accessing builtins such as block height, or creating/calling other contracts. Since the functionality is mocked, the test can do targeted introspection to see if the correct function was called, or walk the heap of the contract working memory to ensure there are no corruptions.

1.31.4 Deploy contract on dev chain

There are some tests in integration which are written in node. These tests start an actual real chain via containers, and then deploying some tests contracts to them and interacting with them.

1.32 Contributing

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

- Consider that users who will read the docs are from different background and cultures and that they have different preferences.
- Avoid potential offensive terms and, for instance, prefer "allow list and deny list" to "white list and black list".
- We believe that we all have a role to play to improve our world, and even if writing inclusive doc might not look like a huge improvement, it's a first step in the right direction.
- We suggest to refer to Microsoft bias free writing guidelines and Google inclusive doc writing guide as starting points.

1.32.1 How to report issues

Please report issues to github issues.

1.32.2 How to contribute code

Code contributions are submitted via pull requests.

Please fork this repository and make desired changes inside a dedicated branch on your fork. Prior to opening a pull request for your branch, make sure that the code in your branch

- does compile without any warnings (run `cargo build --workspace`)
- does not produce any clippy lints (run `cargo clippy --workspace`)
- does pass all unit tests (run `cargo test --workspace`)
- has no merge conflicts with the `main` branch
- is correctly formatted (run `cargo fmt --all` if your IDE does not do that automatically)

1.32.3 Target Specific

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

1.32.4 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.

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.

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.

1.32.5 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.