The Solidity Programming Language

Gabriela Vargas Dalia Camacho

February 2019

Contents

Chapter 1

About Solidity

The Solidity programming language is a Turing complete programming language whose main objective is to create smart contracts that will lead to ether[1] transactions that will be registered on the Ethereum[2] blockchain. Therefore Solidity is considered as a contract-oriented programming language. In order to run a Solidity program and thus proceed with the corresponding transaction, this transaction must be selected by the miners and it adds a new block to the blockchain in every node that contains a copy of the blockchain. Since mining is a costly process, the Solidity contract must specify an amount of gas, in other words the amount of ether that will be paid to the miners if the transaction is selected. The amount of gas necessary to proceed with the transaction will depend on the amount of memory required to store the contract and the amount of instructions that need to be executed. If the transaction runs out of gas before being completed it will not proceed, but the miners will be paid the gas used. The gas restrictions are meant to eliminate the possibility of a program entering a never ending loop. Taking this into account, it may not be the best approach to deal with scientific computation or database administration, because of the elevated cost.

Within this manuscript we describe the most important features of Solidity. In the second Chapter we describe how to begin a Solidity program. In the third Chapter we talk about the contract and the functions and variables that may be included within the contract. In chapter four we go deeply into the supported data types and conversions between them as well as related topics such as polymorphism. Chapter five deals with parameter passing methods, In Chapter 6 we describe how assembly language can be used within a solidity program. In Chapter 7 we explain how run time errors can be treated. In Chapter 8 we review the most important style guidelines. Finally in chapter 9 we give an overall evaluation of Solidity language in terms of its characteristics. We further include three appendices, Appendix A contains the grammar of

[1] Ether corresponds to the cryptocurrency used in Ethereum.

[2] The term Ethereum is used for the blockchain and for the Ethereum Virtual Machine (EVM)

Solidity in the Extended Backus Naur Form. Appendix B includes all opCodes that can be used when assembly language is used, and Appendix C includes a brief guide to the Remix IDE in order that the reader is able to perform the examples here provided.

Chapter 2

How to begin a Solidity program?

Solidity has several releases and is still in the development phase. Different versions handle operations and datatypes in different manners, the compiler also changes. The latest release at this moment is pragma[1] 0.5.5. To define the release we are working with we must define the pragma in the first line of the program.

```
1 pragma solidity ^0.5.2;
```

The previous notation indicates that the compiler will only consider versions starting from 0.5.2 and before 0.6.0. Another way to express this would be as follows:

```
1 pragma solidity >=0.5.2 <0.6.0;
```

If you are interested in the Extended Backus-Naur Form (EBNF) the pragma is constructed as follows:

PragmaDirective = 'pragma' Identifier ([;]+) ';'

where Identifier is defined as :

Identifier = [a-zA-Z_][$a - zA - Z$_0-9]*

Once the pragma has been defined one can either import other contracts or begin with a new contract if the first is not required. To import another contract we can do the following:

```
1 pragma solidity >=0.5.2 <0.6.0;
2 import "other_contract.sol";
```

[1] The term pragma is equivalent to the release version.

However, this will import the names within "other_contract.sol" and they may collide with names meant for our new contract. To avoid ambiguity and collision a better approach is:

```
1 pragma solidity >=0.5.2 <0.6.0;
2 import * as newName from "other_contract.sol";
```

If the latter is done, access to items within "other_contract.sol" will be done as `newName.item1`, where item1 belongs to "other_contract.sol".

One can also import some of the items from another contract without importing the whole contract this can be done as

```
1 pragma solidity >=0.5.2 <0.6.0;
2 import {item1 as otheritem1, item2} from "another\_contract.sol";
```

The item names within the braces are the items imported to our contract, using the word 'as' implies the item will be renamed to avoid collision.

In EBNF import is defined as follows:

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
| 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
| 'import' '{' Identifier ('as' Identifier)? (',' Identifier ('as' Identifier)?)* '}' 'from' StringLiteral ';'

You can find the EBNF for Identifier and for StringLiteral in Appendix A.

You may have noticed we end every line with a semicolon ";". Most expressions in solidity must end with a semicolon.

Once the pragma and the imports have been established we can begin the contract, which contains all the functions that will be executed when the program becomes part of the blockchain. A deeper explanation on contracts is given in Chapter 3.

Chapter 3

The contract

The contract is the core of a Solidity program. Everything but the pragma and imports is specified within the contract. This includes state variables and functions including programmer defined structs and events. The contract may include an optional constructor which will initialise the contract before deployment in the blockchain. Solidity allows for a single constructor, thus overloading is not supported and no optional parameters can be considered within a contract.

To define a contract we use the following

```
1 pragma Solidity >=0.5.2 <0.6.0;
2 contract MyContract{
3     //This is a comment
4     // Here the constructor, state values, functions and events are
          defined
5 }
```

In EBNF
ContractDefinition = ('contract' | 'library' | 'interface') Identifier ('is' InheritanceSpecifier (',' InheritanceSpecifier)*)? '{' ContractPart* '}'

Where InheritanceSpecifier is defined as
InheritanceSpecifier = UserDefinedTypeName ('(' Expression (',' Expression)* ')')?
and contract part is defined as

ContractPart = StateVariableDeclaration | UsingForDeclaration | StructDefinition | ModifierDefinition | FunctionDefinition | EventDefinition | EnumDefinition

The EBNF for StructDefinition, ModifierDefinition are described in Appendix A.

Contracts can inherit functions and state variables from other contracts. If contract B inherits from contract A, then B is derived from A and A is a

parent of B. Derived contracts can be used to preserve readability by dividing the contract into several smaller contracts and defining a final contract that inherits all functions and state variables defined in those smaller contracts.

Consider the following example for inheritance in contracts.

```
pragma solidity ^0.5.0;


contract A{
    uint xa=8;
    function BytestoBits(uint b) public returns(uint){
        return xa*b;
    }
}


contract B{
    uint xb=1024;
    function KBtoBytes(uint b) public returns(uint){
        return xb*b;
    }
}


contract C{
    uint xc=1024;
    function MBtoKB(uint b) public returns(uint){
        return xc*b;
    }
}


contract Conversions is A, B, C{
    function MBtoBits (uint b) public returns(uint){
        return BytestoBits(KBtoBytes(MBtoKB(b)));
    }

    function MBtoBytes (uint b) public returns(uint){
        return KBtoBytes(MBtoKB(b));
    }
}
```

Contract Conversions inherits contracts A, B, and C, thus it can use all functions and state variables from A, B, and C. Moreover, when deploying contract Conversions all public functions can be used externally.

3.1 Constructors

Constructors are used to initialise contracts' state variables before deploying the code into the blockchain. Constructors are optional, but they may be useful to initialise state variables when these require functions, because the initialisation phase using the constructor does not consume any gas. Moreover, the deployed contract does not contain internal functions within the constructor used to initialise the contract.

3.2 State Variables

State variables are stored in the contract storage, this means that they are written in the blockchain. To define a state variable we must indicate the datatype and the name of the variable. The variable can be initialised with a value at the same time. As an example consider defining the state variable myvar that corresponding to an integer, and myvar2 that is also an integer initialised in 20.

```
pragma Solidity >=0.5.2 <0.6.0;
contract MyContract{
    //Initialise state variables
    uint myvar;
    uint myvar2 = 20;
}
```

The EBNF that corresponds to define state variables is the following:

StateVariableDeclaration = TypeName ('public' | 'internal' | 'private' | 'constant')* Identifier ('=' Expression)? ';'

Typename indicates the data type, while public, internal, private, and constant correspond to possible modifiers. More on datatypes can be seen here.

3.3 Functions

Functions in Solidity receive a set of parameters whose data type is defined at the same time the function is defined. By convention the name of the parameters begin with an underscore to enhance readability. When a function is defined the user must also indicate whether the function returns a value and the corresponding datatype. An interesting feature of Solidity functions is that they can have multiple returns. Consider the following example, where a function receives two numbers and returns the sum and the multiplication.

```
    pragma solidity >=0.5.2 <0.6.0;
contract MyContract{
    //Initialise state variables
    uint myvar = 2;
    uint myvar2 = 20;
    function myFunction(uint _a, uint _b) returns(uint sum, uint
        mult){
        uint sum = _a +_b;
        uint mult = _a*_b;
        return(sum, mult);
    }

}
```

In contrast to constructors, there can be an overloading of functions, these means two functions can have the same name and different input parameters.

The EBNF to define functions in Solidity is the following:

FunctionDefinition = 'function' Identifier? ParameterList (ModifierInvocation | StateMutability | 'external' | 'public' | 'internal' | 'private')*('returns' ParameterList)? (';' | Block)

where Block is defined as:

Block = '{' Statement* '}'

For more information on how ParameterList, ModifierInvocation, StateMutability, and Statement are defined, the complete EBNF for Solidity is given in Appendix A.

3.3.1 Function types

Just as there are different data types, functions are considered to be of different types. The scope of a function is determined by the type specified. There are four main types which are `public`, `external`, `internal`, and `private`. These specify the context in which the function can be used, the default is public when not specified otherwise. The characteristics of each type are given in Table 3.1

Type	Description
`public`	Public functions can be called within the contract it was created and from any other contract.
`external`	External functions can be called from other contracts, to call it internally[1] one must specify the function belongs to that specific contract by using `this.` before the function's name.
`internal`	Internal functions can only be called within the contract they were created or derived contracts[2].
`private`	Private functions can only be called within the contract they were specified, but not from derived contracts.

Table 3.1: Function Types

Recall the example used to show how inheritance works. If we change the function type from `public to` internal in contracts A, B, and C we will be able to execute the public functions defined in the contract Conversions, but we won't be able to run the defined in A, B, or C. However if we define any function from A, B, or C as private, the contract Conversions will not compile, because it cannot inherit private functions.

Functions can also use the modifiers `view` and `pure`. The modifier `view` indicates state variables are only accessed, but that no state variables are defined or changed. The modifier `pure` is used when no state variables are accessed nor defined. These specifications allow the programmer to save gas.

[2]Derived contracts are used to achieve high modularity and inherit functions and state values of the parent contract

Consider the following example in which we calculate the area of a circle and the area of a rectangle. For the circle we define π[3] as a state variable, then the function to calculate the area of a circle must be `view`, because π is needed to calculate the area. However, to calculate the area of a rectangle we only need height and width which are provided externally, thus this function is defined as `pure`.

```
pragma solidity ^0.5.0;

contract ExamplePureView{
    int pi = 314159265;

    function circArea(int radius) public view returns (int){
     return pi*radius*radius/(10**6);
    }

    function rectArea(int height, int width)public pure returns(int
        ){
        return height*width;
    }
}
```

Such as data types, some function types can be converted from one type to another. Solidity supports the following conversions[4]:

- Pure functions can be converted to view and non-payable functions.
- View functions can be converted to non-payable functions.
- Payable functions can be converted to non-payable functions.

The `payable` modifier indicates that by executing that function the sender of the function can receive Ether[5]. This modifier is what makes Ether transactions possible.

3.3.2 Modifier functions

Modifier functions are used to create new modifiers for functions. These new modifiers verify certain conditions are met before executing a function. Consider the next example in which we verify $y \neq 0$ before calculating x/y

```
pragma solidity ^0.5.0;

contract ModifierExample{

    modifier YNotZero (int y){
        require(y!=0, "Y must be different to zero");
        _;
    }
```

[3]Solidity does not support doubles nor the `ufixed` type yet, therefore we define it as an `int` and then take some considerations depending on the decimal numbers required.

[4]Possible conversions were taken from https://solidity.readthedocs.io/en/v0.5.3/types.html

[5]Remember Ether is the crypto currency used in Ethereum.

```
9
10      function division(int x, int y) public YNotZero(y) returns(int)
            {
11          return x/y;
12      }
13  }
```

Notices after the require we use '_;' to continue with the function evaluation.

3.3.3 Fallback functions

A fall back function has no name, does not receive any arguments, does not return an output, must be payable and external and it is used whenever the contract receives Ether. At most a contract has one fallback function.

3.4 Events

Events are used in a contract to keep track of what happens to the contract. One can have an event when a specific function is triggered and changes to state variables will be recorded. This information is stored in a log. To indicate an event one must use the `emit` command. Consider the following example from the crypto zombies tutorial https://cryptozombies.io/en/lesson/1/chapter/13.

```
1   pragma solidity >=0.4.22 <0.6.0;
2
3   contract EventsExample{
4       // declare the event
5       event IntegersAdded(uint x, uint y, uint result);
6
7       function add(uint _x, uint _y) public returns(uint) {
8       uint result = _x + _y;
9       // fire an event to let the app know the function was called:
10      emit IntegersAdded(_x, _y, result);
11      return result;
12      }
13  }
```

When the function is executed the log returns the value of x, the value of y, and the result.

Chapter 4

Data Types and Related Topics

4.1 Data Types

Solidity has support for elementary data types, arrays of elementary data types, and structs that are user defined data types. First we will review the elementary data types with their corresponding operations. Next we will explain how to treat arrays and how to define structs.

4.1.1 Elementary data types

The elementary data types are booleans, integers, fixed-point numbers, bytes, strings, addresses, and enums.

Functions are said to be of different data types, but this is addressed in the section dedicated to function modifiers.

Booleans

In Solidity, logical values are given by the `bool` type as in other languages and its valid values are constants `true` and `false`. The operations that can be applied to the boolean type are shown in Table 4.1

Characteristic	**Description**
Name of data type	Boolean
Keyword	`bool`
Valid Values	true, false
Operators/functions	Conjunction (&&) Disjunction (||) Negation (!) Equality (==) Disparity (!=)
Example	`bool activeUser=true;`

Table 4.1: Operations supported by boolean type.

Operators && and || apply the same rules of logical circuits, as the right-hand side operand will only be evaluated if the result of the operation can't be deduced from the value of the left-hand side operand. For example, in the expression f(x)||g(x), the operand g(x) won't be evaluated if the value of f(x) is true.

Integers

Integer numbers are defined as shown in Table 4.2.

Characteristic	**Description**
Name of data type	Signed integer or unsigned integer.
Keyword	`int` - signed integer; 255 bits are used to represent the value and 1 bit corresponds to the sign. `uint` - unsigned integers including 0 and positive numbers; all 256 bits are used to represent the value.
Valid Values	Positive and negative integers between 8 to 256 bits.
Comparison operators	Less than < Less than or equal to <= Greater than > Greater than or equal to >= Equal to == Not equal to ! =
Bitwise operators	And & Or | Xor ^ Not
Unary Arithmetic operators	Positive + Negative -
Binary Arithmetic operators	Addition + Subtraction - Multiplication * Quotient / Remainder % Exponentiation ** Shift left << Shift right >>
Example	`uint16 activeClients = 20000;`

Table 4.2: Operations supported by integer type.

Types `uint` and `int` go up in increments of 8 from the lowest value of 8 to the maximum value of 256 e.g. `int8`, `int16`, `uint16`. Types `uint` and `int` by default are aliases for `uint256` and `int256`, respectively. However, the programmer needs to keep in mind that storage is the most expensive operation in Ethereum and small variables should be used whenever possible.

Other considerations for integer values in Solidity are:

- Division and module by zero throw a runtime exception.
- Shifting numbers by a negative amount throws a runtime exception since the value of x<<y is equivalent to x*2**y and the value of x>>y is equivalent to x/2**y.

The following contains examples of basic operations for integer and boolean variables in Solidity.

```
pragma solidity >=0.4.22 <0.6.0;

contract Calculator {

    function and(bool x, bool y) public pure returns (bool) {
        return (x && y);
    }

    function or(bool x, bool y) public pure returns (bool) {
        return (x || y);
    }

    function addition(int x, int y) public pure returns (int) {
        return (x + y);
    }

    function subtraction(int x, int y) public pure returns (int) {
        return (x - y);
    }

    function greaterThan(int x, int y) public pure returns (int) {
        if (x > y)
            return x;
        else
            return y;
    }

    function greaterThan2(int x, int y) public pure returns (bool)
        {
        return x>y;
    }

    function shiftL (int x, int y) public pure returns (int){
        return (x<<y);
    }
}
```

Fixed Point Numbers

Floating point numbers are not fully supported by Solidity, as they can be declared but cannot be assigned to variables. Similar to integer variables, fixed point numbers can take values with and without a sign. They can be defined as `ufixedMxN` and `fixedMxN` where M defines the total number of bits and can be between 8 and 256, in multiples of 8; and N, which must be between 0 and 80 inclusive and defines the number of bits dedicated to the decimal part. the names `ufixed` and `fixed` are aliases of `ufixed128x18` and `fixed128x18` respectively.

The operations available for fixed point numbers are shown in Table 4.3.

Characteristic	**Description**
Name of data type	Signed fixed point number or unsigned fixed point number.
Keyword	`fixedMxN` - signed fixed point number; 255 bits are used to represent the value and 1 bit corresponds to the sign. `ufixedMxN` - unsigned fixed point number including 0 and positive numbers; all 256 bits are used to represent the value.
Valid Values	Supports values in which M is divisible by 8 and takes values in the interval [8,256]; N must take values between 0 and 80, inclusive.
Comparison operators	Less than < Less than or equal to <= Greater than > Greater than or equal to >= Equal to == Not equal to ! =
Unary Arithmetic operators	Positive + Negative -
Binary Arithmetic operators	Addition + Subtraction - Multiplication * Quotient / Remainder %
Example	fixed a = 3.14

Table 4.3: Operations supported by fixed number type.

Solidity's fixed point variables are different from Java's floating point type, as the number of bits used for the integer part and the fractional part of a fixed point number is specified in variable declaration and must remain the same across the program, while the number of decimals in float numbers can vary at any stage of execution.

Due to the restrictions to operate fixed point numbers in Solidity, programmers have started to use integers instead, especially when working with currencies. For example, to represent $10.25, they would write 1025, with the convention of reserving the 2 or 3 rightmost digits for decimals.

Moreover, Ethereum enthusiasts such as fintech startup Bankex have started initiatives to develop a floating point number implementation using available functions and types such as `uint256`, `bytes32`, additions, multiplications, and bit shifts.

As an example consider calculating the area of a circle which is π^2, where $pi \approx 3.14159265$. If `ufixed` were supported we could have the following code.

```
pragma solidity ^0.5.0;

contract CircleArea{
    ufixed pi = 3.14159265;
    function circArea(ufixed radius) public view returns (ufixed){
       return pi*radius*radius;
    }
}
```

However we must calculate it as follows considering we want a two digit precision and that the radius is in fact an integer.

```
 pragma solidity ^0.5.0;

contract CircleArea{
    int pi = 314159265;

    function circArea(int radius) public view returns (int){
     return pi*radius*radius/(10**6);
    }
}
```

If we were to calculate the area of a circle with radius one we would have an output of 314, considering the two decimal places this corresponds to 3.14.

Bytes

This type is used to store raw byte data. Variables are declared as BytesN, where N must be a number between 1 and 32, inclusive. `byte` is an alias for `bytes1` which is an array of 1 byte. The maximum bytes array is `bytes32`.

The byte operations supported are shown in Table 4.4

Characteristic	Description
Name of data type	Bytes
Keyword	`byte` - 1-byte array. `bytesN` - N-byte array.
Valid Values	Bytes of N arrays where N must be in the interval [1,32].
Comparison Operators	Less than < Less than or equal to <= Greater than > Greater than or equal to >= Equal to == Not equal to ! =
Bitwise operators	And & Or | Xor ˆ Not Shift left << Shift right >>
Indexed access	If x is a bytesN variable, then x[k] for 0<=k<N returns the kth byte (read-only).
Array length	The property .length returns the fixed size of a bytes array.
Example	bytes data = 0x3333;

Table 4.4: Operations supported by bytes type.

Strings

Solidity accepts strings written either with double quotes or single quotes.

While it is relatively easy to operate strings in Java, more coding is required to do the same operations in Solidity. These operations are shown in Table 4.5

Characteristic	Description
Name of data type	String
Keyword	`string`
Valid Values	Arbitrary-length text enconded in UTF-8.
Operators/functions	Not available. Cast to bytes is required.
Example	`string public message;`

Table 4.5: Operations supported by string type.

We provide three examples showing how to operate with strings, even though there are no direct operations on strings.

Example 1: Comparing Strings and Bytes

Strings are similar to bytes arrays, though they don't enable indexed access

to character positions and they don't provide a function to obtain the length of the variable. If the string length can be fixed to a specific number of characters and this number does not change across the contract, it is easier and even cheaper to use bytes arrays instead.

```
pragma solidity >=0.4.22 <0.6.0;

contract BytesOrStrings {
  string constant _string = "Luis Alpizar";
  bytes32 constant _bytes = "Luis Alpizar";
  string public Clientname;

  function  getAsString () public pure returns(string memory) {
    return _string;
  }

  function  getAsBytes() public pure returns(bytes32) {
    return _bytes;
  }

  function setName(string memory newName) public {
        Clientname = newName;

  }
}
```

Example 2: Concatenation

Strings in Solidity are handled as arrays of bytes and will consequently be operated in the same fashion. Therefore, to concatenate two strings, we need to cast them into bytes.

```
pragma solidity >=0.4.22 <0.6.0;

contract Concatenation {

    function strConcat(string memory _a, string memory _b) public
        pure returns (string memory){
        bytes memory _aa = bytes(_a);
        bytes memory _bb = bytes(_b);
        bytes memory ab = new bytes(_aa.length + _bb.length);
        uint k = 0;
        uint i = 0;
        for (i = 0; i < _aa.length; i++) ab[k++] = _aa[i];
        for (i = 0; i < _bb.length; i++) ab[k++] = _bb[i];
        return string(ab);
    }

}
```

Example 3: String Comparison

To compare strings in Solidity we require to perform the following three steps:

1. Cast the strings into bytes.
2. Compare their lengths to save additional steps if the strings don't have the same amount of characters.

3. If the bytes variables don't have the same length, return false as they are different. In other case, apply the built-in cryptographic function for Ethereum `keccak256` to compare if the two inputs are equal.

```
pragma solidity >=0.4.22 <0.6.0;

contract StringComparison {

    function compareStrings (string memory _a, string memory _b)
        public pure returns (bool){
        bytes memory _aa = bytes(_a);
        bytes memory _bb = bytes(_b);

        if(_aa.length != _bb.length) {
        return false;
        } else {
        return keccak256(_aa) == keccak256(_bb);
        }
    }

}
```

Adresses

The type `address` is a special data type that holds a 20-byte value, which is the size of an Ethereum address. This data type works as a base for all contracts, due to Ethereum's primary purpose of facilitating general transactions between agents.

It is possible to query the balance of an address using the property 'balance' and to send Ether, in wei units, to an address using the 'transfer' function. Wei is the smallest denomination on the Ethereum network (1 Ether = 1x10$\hat{1}$8 wei).

The operations available for addresses and more information is given in Table 4.6

Characteristic	**Description**
Name of data type	Address
Keyword	`address`
Valid Values	20-byte Ethereum addresses
Comparison Operators	Less than < Less than or equal to <= Greater than > Greater than or equal to >= Equal to == Not equal to !=
Example	`address public constant 0xabD98F15aCafB9233307fBe7608AADF134E331af;`

Table 4.6: Operations supported by address type.

Consider the following example of balance checking for a single address as well as a function for fund transference between two addresses.

```
pragma solidity >=0.4.22 <0.6.0;

contract Address {
    address public owner;

    function getBalance(address _address) public view returns (
        uint256) {
        return _address.balance;
    }

    constructor() public payable {
        owner = msg.sender;
    }

    function transfer(address payable _to, uint256 _amount)
        external payable{
        require(msg.sender==owner);
        _to.transfer(_amount);
    }
}
```

Enums

Enums are one way to create a user-defined type in Solidity where a keyword can be mapped to an integer value. As an example consider the following program.

```
contract Enums {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight
        ;

    function setSitStill() {
        choice = ActionChoices.SitStill;
    }
}
```

4.1.2 Arrays

Arrays can either be fixed-sized or dinamically-sized. While fixed-sized arrays can be stored either in memory or storage, dinamically-sized arrays can only be stored in the storage. Arrays have a function *length* which returns the size of the array and dynamic storage arrays can be re-sized by adjusting their lenght. Furthermore, dynamic arrays also have a function *push* to add new elements.

The following contract shows different array types supported by solidity.[1]

```
pragma solidity >=0.4.22 <0.6.0;

contract Arrays {
```

[1] Example taken from [1]

```
    //Fixed size array where each element is of type uint
    uint[5] a;
    //Dinamically sized array where each element is of type uint
    uint [] b;
    //Two-dimensional array
    uint8 [10][10] c;
    //Multi-dimensional array having dynamic elements
    uint [][5] d;

    function add (uint _p) public {
        b.push(_p);
    }

    function arrayLength () public view returns (uint){
        return b.length;
    }
}
```

4.1.3 Mappings

Mapping types are used to store key-value pairs. The key can have any type except for mapping, dynamically sized array, an enum and a struct.

```
pragma solidity >=0.4.22 <0.6.0;

contract Mappings {

    struct Seller{
        address addr;
        uint amount;
    }

    mapping(uint => Seller) sellers;

    Seller s= sellers [0];

}
```

4.1.4 Structs

Structs are user defined data types that consist of combining different data types. As an example we can define a struct 'Client' that consists on the client's name as a string, the client's account also as a string, and the money the client has in his/her account as an integer. We can also define an array of structs, in our example we have the array 'clients' whose elements correspond type 'Client' which is a struct.

```
pragma solidity ^0.5.1;

contract BankClients {

    event NewClient(string  name, string account, int money);

    struct Client {
```

```
8           string name;
9           string account;  // Coefficients.
10          int money;
11      }
12
13      Client[] public clients;
14
15    function  createClient(string memory _name, string memory
          _account, int  _money) public{
16        clients.push(Client(_name, _account, _money));
17        emit NewClient(_name, _account, _money);
18    }
19  }
```

4.2 Type conversions

Operations in Solidity require the operands to be of the same data type. Given that Solidity belongs to the category of compiled programming languages, type checking is performed by the compiler and it will try to automatically convert variable types that seem to be different. However, in cases when implicit type conversion is not allowed, there is a group of functions that enable explicit conversions.

Implicit Conversion - Includes cases where no information is lost after the conversion. For example, *uint8* can be converted to *uint16* and *uint128* can shift to *int256*, but *int8* cannot be converted to *uint256* because the unsigned number type does not support negative numbers.

Explicit Conversion - Although explicit type conversion may cause unexpected errors, some popular casting functions area available in Solidity and most of them are used to convert integer sub-types and string variables to bytes variables.

Example: The following function converts an *int8* variable to *uint256*. While conversion of a positive 8-bit number returns a positive 256-bit number, conversion of a negative number into its positive counterpart is not possible with explicit casting functions.

```
1  pragma solidity >=0.4.22 <0.6.0;
2
3  contract TypeConversion {
4      function conversion (int8 _x) public pure returns (uint){
5          uint y = uint(_x);
6          return y;
7      }
8
9  }
```

Furthermore, to facilitate string manipulation in Solidity, casting functions to bytes are available, as shown in 4.1.1.

4.3 Inheritance and Polymorphism

Similar to object-oriented programming languages, Solidity supports contract inheritance and polymorphism to encourage modularity between contracts. The base contract is called *parent contract* and the inheriting contract is called *child contract*. The keyword used to indicate inheritance from other class is the word *is*

Example 1 The following code is an example of inheritance implementation in Solidity. It involves a parent contract called "Messages" and a child contract called called "Message1", which will have the capability to invoke the parent function as well as its own function.

```
pragma solidity >=0.4.22 <0.6.0;
contract Messages{
    function catchphrase () public pure returns(string memory){
        return "This is a simple contract";
    }
}

contract Message1 is Messages{
    function catchphrase () public pure returns(string memory){
        return "This contract will transfer money from account X to
             account Y";
    }
}
```

Example 2 Child contracts not only inherit functions from their parents, they also inherit state and local variables, as shown in the example below:

```
pragma solidity >=0.4.22 <0.6.0;

contract MessageP{
    string public phrase = "This is a simple contract";
}

contract MessageC is MessageP{
    function catchphrase () public view returns(string memory){
        return phrase;
    }
}
```

4.4 Strongly-typed or Weakly-typed

Solidity is a statically-typed language which means that type checking is performed during translation, in contrast to dynamically-typed languages that perform type checking during execution time (JavaScript, for example). This implies that Solidity is strongly-typed compared to Javascript.

Chapter 5

Parameter Passing methods

Data types in Solidity can be categorised into two groups according to their parameter passing approach.

Call-by-value - The parameter's value is copied to the stack at beginning of program execution. This condition implies that any changes applied to this variables are limited to the local environment of the function and they disappear at the end of execution.

This category includes the following data types:

1. Boolean
2. Integer
3. Fixed Point Numbers
4. Addresses
5. BytesN

Call-by-reference - This type of parameters are more complex than the call-by-value types and they often occupy more than 256 bits of memory. They should be operated carefully as some operations can produce a significant waste in gas.

Data types included in this category are listed below:

1. Arrays
2. Structs
3. Mappings

5.1 Binding and scope

Solidity considers two different locations to store variables: storage and memory. The keyword **storage** refers to variables stored permanently on the blockchain and the keyword **memory** refers to temporary variables erased after the function execution. The default value for function parameters is memory, and storage is the default value for local variables. State variables must have storage location.

Chapter 6

Using assembly language in Solidity

A distinctive aspect of Solidity is the support it has for coding in assembly language. It may seem counter intuitive to write in a low level language within a high level language such as Solidity. However, writing code in assembly language diminishes the cost of translation, and thus decrements the gas needed to run the code.

Assembly language can be used in Solidity in two manners. The first one consists on combining Solidity language with assembly language within the same program. This is called **inline** assembly. The second option is to write the whole program in assembly language, this is known as **standalone** assembly.

The instruction `assembly{}` indicates where the assembly code starts. The possible instructions of the assembly code correspond to the possible opcodes. The list of supported opcodes is provided in Appendix X.

To enhance readability it is suggested that the programmer writes the opcodes in a functional style for example[1]:

```
1 mstore(0x80, add(mload(0x80), 3))
```

instead of

```
1 3 0x80 mload add 0x80 mstore
```

When using assembly code the programmer must be very careful when accessing or storing in data locations. Free memory begins in slot 0x40 and the programmer can start storing variables from that slot onwards. However, there is no guarantee that the memory is empty beforehand. To access the memory slot of a variable `x` previously defined, use the command `x_slot`.

To ease writing complex assembly code, for loops[2] and conditional state-

[1] Example taken from https://solidity.readthedocs.io/en/latest/assembly.html

[2] While loops can be written as special cases of for loops.

ments such as the if statement[3] and the switch statement are supported. This was done in order to avoid defining jumps explicitly.

To show the cost benefits of writing code in assembly language we consider the example given in https://solidity.readthedocs.io/en/latest/assembly.html in which the sum of the elements from an array is calculated using solidity code, inline assembly language, and standalone solidity code.

```
pragma solidity >=0.4.16 <0.6.0;

library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := add(o_sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory _data) public pure returns (uint o_sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(_data)

            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing _data would result in an unusable
            //       _data variable after this assembly block
            let data := add(_data, 0x20)

            // Iterate until the bound is not met.
            for
```

[3] For assembly language in Solidity if statements do not allow for an else statement, however the switch statement can be used for that purpose.

```
38                  { let end := add(data, mul(len, 0x20)) }
39                  lt(data, end)
40                  { data := add(data, 0x20) }
41              {
42                  o_sum := add(o_sum, mload(data))
43              }
44          }
45      }
46  }
```

The gas costs for an array that contained the values from one to ten using the Remix IDE are presented in Table 6.1

Method	**Transaction Cost**	**Execution Cost**
Solidity	25,560 gas	1,984 gas
Inline Assembly	25,158 gas	1,582 gas
Standalone Assembly	24,994 gas	1,418 gas

Table 6.1: Gas costs of adding the elements of a vector containing values from one to ten using different coding approaches using the Remix IDE.

Chapter 7

Runtime Errors

Exception errors occur when some given conditions are violated and this cannot be foreseen before executing the code. In Solidity when an exception error occurs all changes to state variables are reverted. Moreover, Solidity has two ways for expressing exception errors in predefined cases and the user may add more conditions that when unmet can have any of the two behaviours.

The first way in which exceptions are treated is known as **assert**-style exceptions. When they occur no specific message about the error is provided. Furthermore this type of errors consume gas until the gas limit is reached. The predefined cases for assert-style exceptions are[1]:

1. Trying to access a negative index on an array, or an index whose value is larger than the length of the array.
2. The latter but for fixed length `bytesN` instead of array.
3. Dividing by zero or trying to get a modulo zero. (4/0 or 4%0)
4. Shift by a negative value.
5. Converting a large or negative value to an `enum`.
6. Calling a zero-initialized variable of internal function type.
7. When the condition within an assert function is false.

The `assert` function is used as follows:

```
1 pragma solidity >=0.4.22 <0.6.0;
2
3 contract BreakIfZero {
4
5     function break(int x) public pure returns(int){
6         assert(x!=0);
```

[1]Cases that lead to assert-style exceptions were taken from https://solidity.readthedocs.io/en/latest/control-structures.html#assert-and-require

```
7           return(x);
8       }
9   }
```

Since assert-style exceptions use all the gas it is only recommended for testing and programs should never reach the assert function.

In contrast to assert-style exceptions, Solidity supports **require**-style exceptions. These provide an output message for the programmer to indicate the problem and gas consumption stops once the exception is reached.

Predefined require-style exceptions are the following[2]:

1. The condition on the requirement argument is false.
2. Calling a function via a message call but it does not finish properly[3].
3. If you create a contract using the `new` keyword but the contract creation does not finish properly.
4. Performing an external function call targeting a contract that contains no code.
5. If your contract receives Ether via a public function without `payable` modifier.
6. If your contract receives Ether via a public getter function.
7. If a `.transfer()` fails.

There are two ways in which the programmer can produce a require-style exception. Either using the `require` function or the `revert` function. The `require` function is similar to the `assert` expression, but one may write an additional message. The `revert` function does not have a conditional statement as an input, it will revert all changes whenever it is called. Consider the following example for both `require` and `revert`.

```
1   pragma solidity >=0.4.22 <0.6.0;
2
3   contract Division{
4
5       function divisionRequire (int x, int y) public pure returns(int
            ){
6           require(y!=0, "Division by zero not allowed");
7           return x/y;
8       }
9
10      function divisionRevert (int x, int y) public pure returns(int)
            {
11
12        if(y==0){
```

[2]These were also taken from https://solidity.readthedocs.io/en/latest/control-structures.html#assert-and-require

[3]Not finishing properly implies running out of gas, having no matching function, or throwing an exception itself

```
13              revert("Division by zero not allowed");
14          }
15            return x/y;
16        }
17
18    }
```

The require or revert functions should be preferred by the programmer for exception errors that may occur when actually running the program because of gas consumption.

Chapter 8

Style Guidelines

Solidity's style guidelines are still changing as Solidity is in constant development. However, following the proposed guidelines makes code more readable. The guideline proposed is based on the PEP8 from Python and all the details can be found in https://solidity.readthedocs.io/en/latest/style-guide.html

The most important aspects are related to blank lines, maximum length, order of layout within the program, and naming styles.

In summary blanklines should only appear between contracts or inside contracts between constructors, functions, and events when these take more than a single line.

The maximum length of a line should be less than 79 or 99 characters. To maintain lines short argument lists should start in the line after declaring the function, constructor, or event. And there must be only one argument per line. As an example consider the following code.

```
1  pragma solidity >=0.4.22 <0.6.0;
2
3  contract MyContract{
4      function getStats (
5          int param1,
6          int param2,
7          int param3,
8          int param4
9      )
10         public pure returns
11     (
12         int sum,
13         int mean,
14         int variance
15     )
16     {
17         sum = param1 + param2 + param3 + param4;
18         mean = sum/4;
19         variance = (param1 - mean)*(param1-mean);
20         variance = variance + (param2 - mean)*(param2-mean);
21         variance = variance + (param3 - mean)*(param3-mean);
```

```
22          variance = variance + (param4 - mean)*(param4-mean);
23          variance = variance/3;
24          return (sum,  mean, variance);
25      }
26
27
28  }
```

The order layout within the contract should be the following:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts
    A. Type declarations
    B. State variables
    C. Events
    D. Functions
        a. Constructor
        b. Fallback function (if exists)
        c. External
            i. Others
            ii. View
            iii. Pure
        d. Public
            i. Others
            ii. View
            iii. Pure
        e. Internal
            i. Others
            ii. View
            iii. Pure
        f. Private
            i. Others
            ii. View
            iii. Pure

Finally names are suggested to be defined according to Table 8.1

Format	Nameable	Example
CapWords	Libraries	`MyLibrary`
	Contract	`MyContract`
	Struct	`MyStruct`
	Events	`MyEvent`
	Enums	MyEnum
mixedCase	Function Names	`fName`
	Function Arguments	`firstArg`
	State Variables	`xVar`
	Modifiers	`myModifier`
UPPER_CASE_WITH_UNDERSCORES	Constants	`CONSTANT_X`

Table 8.1: Suggested format for nameables.

The formats provided in Table 8.1 correspond to the latest version. For `pragma ^ 0.4.25;` it was reccommended to begin function argument names with an underscore.[1]

[1] This convention is given in https://cryptozombies.io/en/lesson/1/chapter/7

Chapter 9

General Evaluation

In this section we evaluate Solidity language in terms of different criteria.

9.1 Expressivity

Solidity has some predefined operators, as seen in Tables 4.1-4.6. However, these operators do not always work and there are not many predefined functions. The examples of calculating the area of a circle and the one that compares two strings show how a lot of tricks and operations are needed to perform simple procedures. Moreover, sometimes predefined operations do not hold. As an example consider once again the area of a circle, now assume the radius is not an integer, and that the user specifies the number of decimals given, and the number of decimals needed for the output. Then we would have the following code, which in theory should work, but it throws the error:
`browser/CircleArea.sol:23:31: TypeError: Operator ** not compatible with types int_const 10 and int256 return pi*radius*radius/(10**decs);`

```
1  pragma solidity ^0.5.0;
2
3  contract CircleArea{
4      int pi = 314159265;
5      int  decPi = 8;
6
7
8      function circ_area2(
9      int radius
10     int  decRadius,
11     int  decOutput
12     )
13     public view returns (int)
14     {
15      int  decs = decPi +  decRadius - decOutput;
16      return pi*radius*radius/(10**decs);
17     }
18 }
```

Another aspect to consider is the lack of double or floating point operations, which leave a lot of work to the programmer when not working with integer values.

9.2 Well-definedness

Solidity is not well defined yet, there are several examples where it is unclear why exception errors occur. Some examples from the Solidity guides in https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html?highlight=coin such as the next fail without any apparent explanation.

```
1  pragma solidity ^0.5.0;
2
3  contract Coin {
4      // The keyword "public" makes those variables
5      // easily readable from outside.
6      address public minter;
7      mapping (address => uint) public balances;
8
9      // Events allow light clients to react to
10     // changes efficiently.
11     event Sent(address from, address to, uint amount);
12
13     // This is the constructor whose code is
14     // run only when the contract is created.
15     constructor() public {
16         minter = msg.sender;
17     }
18
19     function mint(address receiver, uint amount) public {
20         require(msg.sender == minter);
21         require(amount < 1e60);
22         balances[receiver] += amount;
23     }
24
25     function send(address receiver, uint amount) public {
26         require(amount <= balances[msg.sender], "Insufficient
               balance.");
27         balances[msg.sender] -= amount;
28         balances[receiver] += amount;
29         emit Sent(msg.sender, receiver, amount);
30     }
31 }
```

Also, examples from previous versions do not compile anymore in the latest pragmas.

Moreover, fixed point numbers although well defined within the documents are not yet suppoted.

9.3 Data types and structures

Solidity has elementary predefined data types, such as address, boolean, string, integers, unsigned integers, bytes, fixed, and unsigned fixed. Moreover, the user can create new data types by using structs or enums. Thus, Solidity is quite flexible in terms of possible data types it supports. However, it is also strongly-typed and the datatype of a variable must always be defined before assigning a value. Some conversions are allowed, but these restricted. For more information on datatypes go to Chapter 4.

9.4 Modularity

Solidity is highly modular, contracts themselves can be seen as modules that contain functions that are also modules. To avoid having long codes one can divide a program into different contracts which are saved in different files. One contract can import another contract as seen in Chapter 2. This provides the possibility to the programmer of a better organisation, and therefore it also helps readability.

9.5 Input-output facilities

In terms of input facilities Solidity supports the input of other Solidity programs, however there is no information on reading data from a different file format. In terms of output facilities we may consider the event functions one can define within the contracts, that will let the user know when a given action is executed. Saving results into a different file does not seem possible, because there is no information on that aspect. The reason for this is that the programs transactions are added to the blockchain and the necessary outputs such as sending or receiving ether are done within the transaction process, but nothing else is needed as an output.

9.6 Portability/standardization

As seen in Chapter 2 Solidity is still in a development phase, therefore new versions are being released, and how the compiler interprets the instructions is still changing. Therefore one must define the pragma before starting the contract. The user must keep this in mind to avoid unexpected results when executing the program. Another aspect to consider is that Ethereum is open source, then Solidity is also open source which could lead to different versions of Solidity. In terms of portability, Solidity is constructed over the Ethereum Virtual Machine, which allows for portability since no further requirements are needed.

9.7 Efficiency

If we run a Solidity program altering the blockchain, the process is not very efficient, because miners have to select it as the winning transaction, and then the program is executed in every node, which is highly resource consuming.

9.8 Pedagogy/simplicity

Solidity as a programming language is very similar to Java and C, therefore users familiar to the latter can easily start programming in Solidity. The tricky part, however, is to understand the role of smart contracts in the overall Ethereum blockchain. For new programmers it may not be that easy to start with Solidity, because it is very structured. However, there are several online sources that can help the learning process. For one the Solidity guide is available online in https://Solidity.readthedocs.io/en/latest/index.html which can be downloaded in different formats. A more practical tool is the CryptoZombies tutorial by the Loom Network, which can be found in https://cryptozombies.io/.

9.9 Generality

Since Solidity is a Turing complete language one could theoretically do everything a computer does in Solidity. However, when considering costs of running a program, which alters the blockchain, is not convenient for usage different to transactions.

9.10 Modeling ability

Just as mentioned in the generality section, even though in theory Solidity could model different real world scenarios, it may not be the best approach, unless we want to model transactions.

9.11 Readability

Because of modularity, Solidity becomes readable. Moreover, there are some clear style guidelines that when they are followed they enhance readability. Solidity has also a readability aid when numeric literals are constructed, because one can separate numbers with an underscore to distinguish better thousands from hundreds and so on. The program may not be as readable if the programmer decides to write part of the code in assembly language, the latter saves gas but it makes it more difficult to understand as a human.

9.12 Semantic clarity

Semantic clarity in programming languages refers to the meaning of the statements and the ambiguity of two programs that are written with different statements, but produce a similar output.

An example of semantic ambiguity in Solidity can be found in *address.transfer(amount)* and *address.send(amount)* functions. Both of them are meant to solve the problem of executing electronic fund transfers between two different Ethereum accounts, but there are subtle differences between them. For instance, they perform different in terms of unsuccessful transfer attempts, since *transfer* throws an exception at execution time but *send* needs to be written inside of a *require* statement in order to avoid unexpected gas withdrawals.

Bibliography

[1] Arshdeep Bahga and Vijay Madisetti. *Blockchain Applications: A Hands-On Approach*. VPT, 2017.

[2] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[3] Ethereum. Solidity documentation release 0.5.5, 21 February 2019.

Appendix A

Solidity EBNF description

The EBNF for solidity was taken from https://github.com/ethereum/solidity/blob/develop/docs/grammar.txt without any modification except from format changes to make it easier to read. Here they use '(...)? to indicate that whatever is inside is optional instead of using '[...]'. And they do not use '{...}' to indicate optional, the use (...)* instead.
The definitions for solidity language are the following.

SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

Pragma actually parses anything up to the trailing ';' to be fully forward-compatible.

PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'| 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';' | 'import' '{' Identifier ('as' Identifier)? (',' Identifier ('as' Identifier)?)* '}' 'from' StringLiteral ';'

ContractDefinition = ('contract' | 'library' | 'interface') Identifier ('is' InheritanceSpecifier (',' InheritanceSpecifier)*)? '{' ContractPart* '}'

ContractPart = StateVariableDeclaration
| UsingForDeclaration
| StructDefinition
| ModifierDefinition
| FunctionDefinition
| EventDefinition
| EnumDefinition

InheritanceSpecifier = UserDefinedTypeName ('(' Expression (',' Expression)* ')')?

StateVariableDeclaration = TypeName ('public' | 'internal' | 'private' | 'constant')* Identifier ('=' Expression)? ';'

UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'

StructDefinition = 'struct' Identifier '{' (VariableDeclaration ';' (VariableDeclaration ';')*) '}'

ModifierDefinition = 'modifier' Identifier ParameterList? Block

ModifierInvocation = Identifier ('(' ExpressionList? ')')?

FunctionDefinition = 'function' Identifier? ParameterList (ModifierInvocation | StateMutability | 'external' | 'public' | 'internal' | 'private')* ('returns' ParameterList)? (';' | Block)

EventDefinition = 'event' Identifier EventParameterList 'anonymous'? ';'

EnumValue = Identifier

EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

ParameterList = '(' (Parameter (',' Parameter)*)? ')'

Parameter = TypeName StorageLocation? Identifier?

EventParameterList = '(' (**EventParameter** (',' EventParameter)*)? ')'

EventParameter = TypeName 'indexed'? Identifier?

FunctionTypeParameterList = '(' (FunctionTypeParameter (',' FunctionTypeParameter)*)? ')'

FunctionTypeParameter = TypeName StorageLocation?

Semantic restriction: mappings and structs (recursively) containing mappings are not allowed in argument lists

VariableDeclaration = TypeName StorageLocation? Identifier

TypeName =
ElementaryTypeName

| UserDefinedTypeName
| Mapping
| ArrayTypeName
| FunctionTypeName
| ('address' 'payable')

UserDefinedTypeName = Identifier ('.' Identifier)*

Mapping = 'mapping' '(' **ElementaryTypeName** '=¿' TypeName ')'

ArrayTypeName = TypeName '[' Expression? ']'

FunctionTypeName = 'function' FunctionTypeParameterList ('internal' | 'external' | StateMutability)* ('returns' FunctionTypeParameterList)?

StorageLocation = 'memory' | 'storage' | 'calldata'

StateMutability = 'pure' | 'view' | 'payable'

Block = '{' Statement* '}'

Statement = IfStatement | WhileStatement | ForStatement | Block | InlineAssemblyStatement | (DoWhileStatement | PlaceholderStatement | Continue | Break | Return | Throw | EmitStatement | SimpleStatement) ';'

ExpressionStatement = Expression

IfStatement = 'if' '(' Expression ')' Statement ('else' Statement)?

WhileStatement = 'while' '(' Expression ')' Statement

PlaceholderStatement = '_'

SimpleStatement = VariableDefinition | ExpressionStatement

ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';' (ExpressionStatement)? ')' Statement

InlineAssemblyStatement = 'assembly' StringLiteral? AssemblyBlock

DoWhileStatement = 'do' Statement 'while' '(' Expression ')'

Continue = 'continue'

Break = 'break'

Return = 'return' Expression?

Throw = 'throw'

EmitStatement = 'emit' FunctionCall

VariableDefinition = (VariableDeclaration | '(' VariableDeclaration? (',' VariableDeclaration?)* ')') ('=' Expression)?

Precedence by order (see github.com/ethereum/solidity/pull/732)

Expression = Expression ('++' | '–')
|NewExpression
| IndexAccess
| MemberAccess
| FunctionCall
| '(' Expression ')'
| ('!' | '~' | 'delete' | '++' | '–' | '+' | '-') Expression
| Expression '**' Expression
| Expression ('*' | '/' | '%') Expression
| Expression ('+' | '-') Expression
| Expression ('<<' | '>>') Expression
| Expression '&' Expression
| Expression '^' Expression
| Expression '|' Expression
| Expression (' <' | '>' | '<=' | '>=') Expression
| Expression ('==' | '!=') Expression
| Expression '&&' Expression
| Expression '| |' Expression
| Expression '?' Expression ':' Expression
| Expression ('=' | '|=' | '^=' | '~=' | '¡¡=' | '>>=' | '+ =' | '− =' | '* =' | '/ =' | '% =') Expression
| PrimaryExpression

PrimaryExpression = BooleanLiteral
| NumberLiteral
| HexLiteral
| StringLiteral
| TupleExpression
| Identifier
| ElementaryTypeNameExpression

ExpressionList = Expression (',' Expression)*

NameValueList = Identifier ':' Expression (',' Identifier ':' Expression)*

FunctionCall = Expression '(' FunctionCallArguments ')'

FunctionCallArguments = '{' NameValueList? '}'
|ExpressionList ?

NewExpression= 'new' TypeName

MemberAccess = Expression '.' Identifier

IndexAccess = Expression '[' Expression? ']'

BooleanLiteral = 'true' | 'false'

NumberLiteral = (HexNumber | DecimalNumber) (' ' NumberUnit)?

NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether' | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'

HexLiteral = 'hex' (''' ([0-9a-fA-F]{2})* ''' | '́ ([0-9a-fA-F]{2})* '́)

StringLiteral = ''' ([∧" ◦ \\] | '\\' .)* '''

Identifier = [a-zA-Z_][$a - zA - Z$0-9]*

HexNumber = '0x' [0-9a-fA-F]+

DecimalNumber = [0-9]+ ('.' [0-9]*)? ([eE] [0-9]+)?

TupleExpression = '(' (Expression? (',' Expression?)*)? ')' | '[' (Expression (',' Expression)*)? ']'

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | Int | Uint | Byte | Fixed | Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' | 'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' | 'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' | 'int248' | 'int256'

Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' | 'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' | 'uint112' |

'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' | 'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' | 'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' | 'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' | 'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' | 'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' | 'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'

Fixed = 'fixed' | ('fixed' [0-9]+ 'x' [0-9]+)

Ufixed = 'ufixed' | ('ufixed' [0-9]+ 'x' [0-9]+)

AssemblyBlock = '{' AssemblyStatement* '}'

AssemblyStatement =
AssemblyBlock
| AssemblyFunctionDefinition
| AssemblyVariableDeclaration
| AssemblyAssignment
| AssemblyIf
| AssemblyExpression
| AssemblySwitch
| AssemblyForLoop
| AssemblyBreakContinue

AssemblyFunctionDefinition = 'function' Identifier '(' AssemblyIdentifierList? ')' ('-¿' AssemblyIdentifierList)? AssemblyBlock

AssemblyVariableDeclaration = 'let'

AssemblyIdentifierList (':=' AssemblyExpression)?

AssemblyAssignment = AssemblyIdentifierList ':=' AssemblyExpression

AssemblyExpression = AssemblyFunctionCall | Identifier | Literal

AssemblyIf = 'if' AssemblyExpression AssemblyBlock

AssemblySwitch = 'switch' AssemblyExpression (AssemblyCase AssemblyDefault? | AssemblyDefault)

AssemblyCase = 'case' Literal AssemblyBlock

AssemblyDefault = 'default' AssemblyBlock

AssemblyForLoop = 'for' AssemblyBlock AssemblyExpression AssemblyBlock AssemblyBlock

AssemblyBreakContinue = 'break' | 'continue'

AssemblyFunctionCall = Identifier '(' (AssemblyExpression (',' AssemblyExpression)*)? ')'

AssemblyIdentifierList = Identifier (',' Identifier)*

Appendix B

OpCodes for assembly language

Here we have all the opcodes that the assembly language takes as valid, what the instructions do and when did each of the instructions appeared according to https://solidity.readthedocs.io/en/latest/assembly.html. No changes were done to the opcodes list

Instruction	Present Since	Explanation
stop	F	stop execution, identical to return(0,0)
add(x, y)	F	x + y
sub(x, y)	F	x - y
mul(x, y)	F	x * y
div(x, y)	F	x / y
sdiv(x, y)	F	x / y, for signed numbers in two's complement
mod(x, y)	F	x % y
smod(x, y)	F	x % y, for signed numbers in two's complement
exp(x, y)	F	x to the power of y
not(x)	F	$\tilde{x}$, every bit of x is negated
lt(x, y)	F	1 if x < y, 0 otherwise
gt(x, y)	F	1 if x > y, 0 otherwise
slt(x, y)	F	1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)	F	1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)	F	1 if x == y, 0 otherwise
iszero(x)	F	1 if x == 0, 0 otherwise
and(x, y)	F	bitwise and of x and y
or(x, y)	F	bitwise or of x and y
xor(x, y)	F	bitwise xor of x and y
byte(n, x)	F	nth byte of x, where the most significant byte is the 0th byte

Instruction	Present Since	Explanation
shl(x, y)	C	logical shift left y by x bits
shr(x, y)	C	logical shift right y by x bits
sar(x, y)	C	arithmetic shift right y by x bits
addmod(x, y, m)	F	(x + y) % m with arbitrary precision arithmetic
mulmod(x, y, m)	F	(x * y) % m with arbitrary precision arithmetic
signextend(i, x)	F	sign extend from (i*8+7)th bit counting from least significant
keccak256(p, n)	F	keccak(mem[p...(p+n)))
jump(label)	F	jump to label / code position
jumpi(label, cond)	F	jump to label if cond is nonzero
pc	F	current position in code
pop(x)	F	remove the element pushed by x
dup1 ... dup16	F	copy nth stack slot to the top (counting from top)
swap1 ... swap16 *	F	swap topmost and nth stack slot below it
mload(p)	F	mem[p...(p+32))
mstore(p, v)	F	mem[p...(p+32)) := v
mstore8(p, v)	F	mem[p] := v & 0xff (only modifies a single byte)
sload(p)	F	storage[p]
sstore(p, v)	F	storage[p] := v

Instruction	Present Since	Explanation
msize	F	size of memory, i.e. largest accessed memory index
gas	F	gas still available to execution
address	F	address of the current contract / execution context
balance(a)	F	wei balance at address a
caller	F	call sender (excluding delegatecall)
callvalue	F	wei sent together with the current call
calldataload(p)	F	call data starting from position p (32 bytes)
calldatasize	F	size of call data in bytes
calldatacopy(t, f, s)	F	copy s bytes from calldata at position f to mem at position t
codesize	F	size of the code of the current contract / execution context
codecopy(t, f, s)	F	copy s bytes from code at position f to mem at position t
extcodesize(a)	F	size of the code at address a
extcodecopy(a, t, f, s)	F	like codecopy(t, f, s) but take code at address a

Instruction	Present Since	Explanation
returndatasize	B	size of the last returndata
returndatacopy(t, f, s)	B	copy s bytes from returndata at position f to mem at position t
extcodehash(a)	C	code hash of address a
create(v, p, n)	F	create new contract with code mem[p... (p+n)) and send v wei and return the new address
create2(v, p, n, s)	C	create new contract with code mem[p... (p+n)) at address keccak256(0xff . this . s . keccak256(mem[p... (p+n))) and send v wei and return the new address, where 0xff is a 8 byte value, this is the current contract's address as a 20 byte value and s is a big-endian 256-bit value
call(g, a, v, in, insize, out, outsize)	F	call contract at address a with input 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
callcode(g, a, v, in, insize, out, outsize)	F	identical to call but only use the code from a and stay in the context of the current contract otherwise
delegatecall(g, a, in, insize, out, outsize)	H	identical to callcode but also keep caller and callvalue
staticcall(g, a, in, insize, out, outsize)	B	identical to call(g, a, 0, in, insize, out, outsize) but do not allow state modifications
return(p, s)	F	end execution, return data mem[p... (p+s))
revert(p, s)	B	end execution, revert state changes, return data mem[p... (p+s))

Instruction	**Present Since**	**Explanation**
selfdestruct(a)	F	end execution, destroy current contract and send funds to a
invalid	F	end execution with invalid instruction
log0(p, s)	F	log without topics and data mem[p. . . (p+s))
log1(p, s, t1)	F	log with topic t1 and data mem[p. . . (p+s))
log2(p, s, t1, t2)	F	log with topics t1, t2 and data mem[p. . . (p+s))
log3(p, s, t1, t2, t3)	F	log with topics t1, t2, t3 and data mem[p. . . (p+s))
log4(p, s, t1, t2, t3, t4)	F	log with topics t1, t2, t3, t4 and data mem[p. . . (p+s))
origin	F	transaction sender
gasprice	F	gas price of the transaction
blockhash(b)	F	hash of block nr b - only for last 256 blocks excluding current
coinbase	F	current mining beneficiary
timestamp	F	timestamp of the current block in seconds since the epoch
number	F	current block number
difficulty	F	difficulty of the current block
gaslimit	F	block gas limit of the current block

Appendix C

A quick guide to start using the Remix IDE

To test Solidity contracts without sending them to the Ethereum blockchain and spending actual gas on them we can use the Remix IDE. It can be accessed online in https://remix.ethereum.org.
The Remix environment looks as shown in Figure C.1 when first entering the online Remix IDE.

Figure C.1: Initial view of the Remix IDE

The left hand side window contains the '.sol' files that have been programmed online. The central upper window shows the solidity program that one is editing or creating. The central lower window is the console, here we can view the results

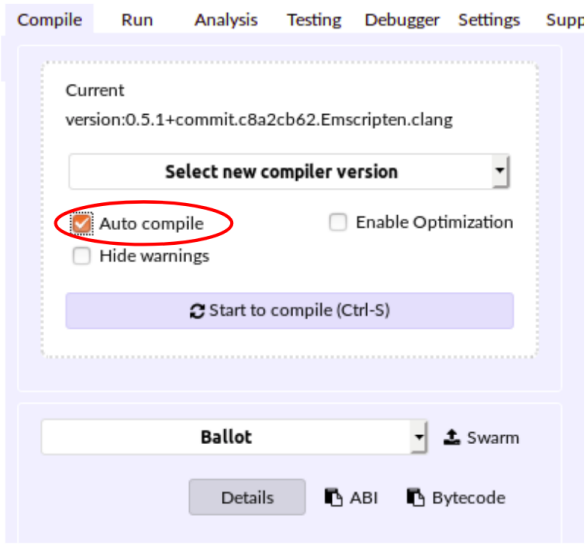

Figure C.2: Compile Window

from running solidity programs. The left hand side window has several tabs, within this guide we will only concentrate in the 'Compile' and 'Run' windows. In the 'Compile' window we indicate the compiler for our program and when should the code be compiled. The most resource efficient is by clicking on 'Start to Compile' whenever we require or we can just select 'Auto compile' and all our programs will be compiled instantly.
The 'Run' window has several elements to consider. The first is the Environment, as shown in Figure C.3 we select the JavaScript VM to execute the codes provided in this document.

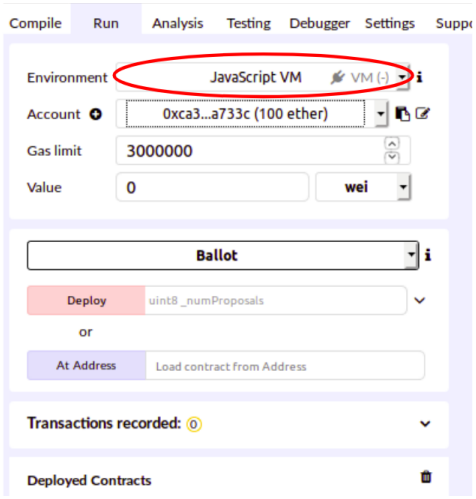

Figure C.3: Environment in 'Run' window.

In the account tab one must give an Ethereum's account address, the Remix IDE provides five default accounts with 100 ether each as shown in Figure C.4.

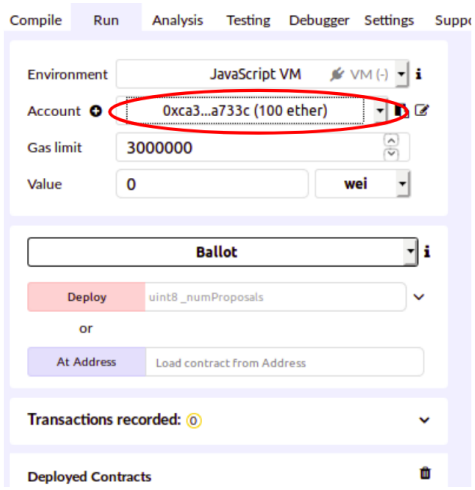

Figure C.4: Account in 'Run' window.

The gas limit must be specified in the 'Gas limit' tab as shown in Figure C.5. Remember that for a transaction to proceed the gas should cover the transaction and execution costs. The examples here provided do not need more gas than the default, but you can increase the gas limit for more costly transactions or decrease it if you are certain about the amount of gas you need.

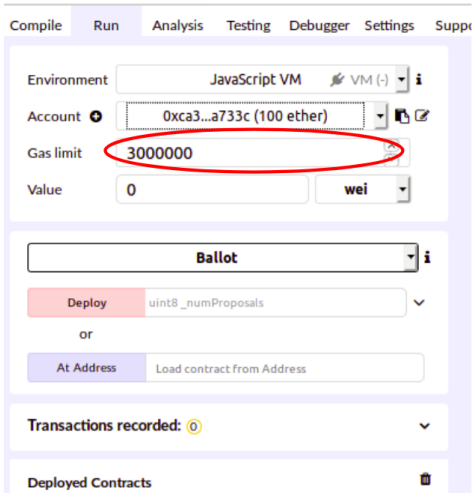

Figure C.5: Gas in 'Run' window.

Once a contract is correctly compiled its name will appear in the 'Run' window, by clicking on 'Deploy' (Figure C.6) the contract is deployed, only after that public and external functions can be executed within the IDE.

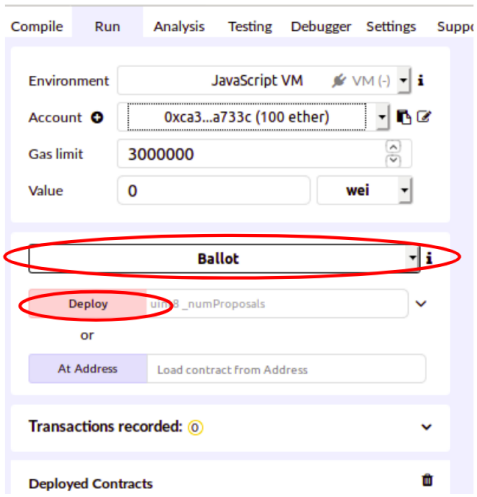

Figure C.6: Deployment in 'Run' window.

Once the contract is deploy we select it by clicking on it and all the public and external functions will be displayed. Each of the functions will have a block where we can type the input arguments. After writing the values we click on the function's name and it will be executed. (See Figure C.7)

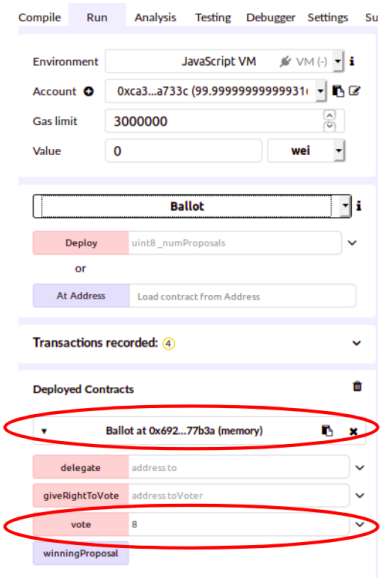

Figure C.7: Execute a function in 'Run' window.

After running the program notice that the amount of ether in the working account is reduced, due to transaction and execution costs. To view the output. we must turn to the console, it will show that a function was executed or it will throw an error message if an exception error occurred. We can see the details by clicking on the arrow next to the 'Debug' button. In Figure C.8 we show the details of running the 'vote' function from the predefined 'Ballot' contract. The total gas used can be monitored in the rows corresponding to transaction cost and execution cost. The output of a function and the log that results from an event. In this specific case the function 'vote' did not return an output, nor emitted an event.

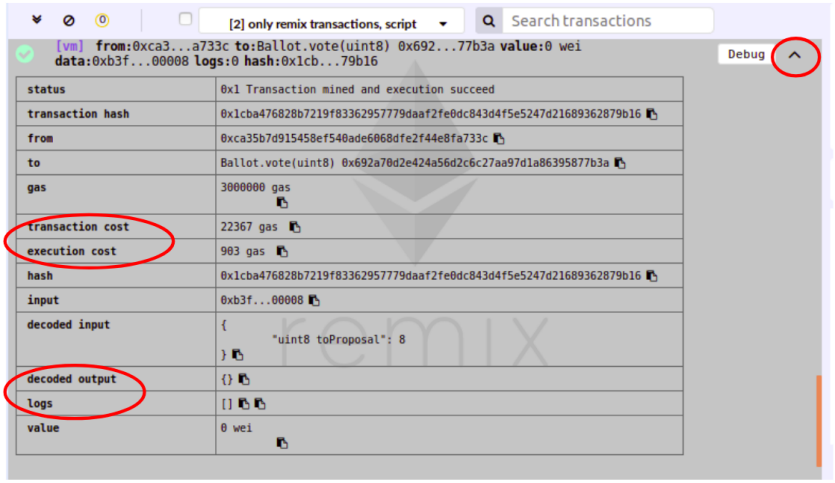

Figure C.8: Information on the execution.