**Yul**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul)

Yul (previously also called JULIA or IULIA) is an intermediate language that can be compiled to bytecode for different backends.

It can be used in stand-alone mode and for “inline assembly” inside Solidity. The compiler uses Yul as an intermediate language in the IR-based code generator (“new codegen” or “IR-based codegen”). Yul is a good target for high-level optimisation stages that can benefit all target platforms equally.

**Motivation and High-level Description**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#motivation-and-high-level-description)

The design of Yul tries to achieve several goals:

1. Programs written in Yul should be readable, even if the code is generated by a compiler from Solidity or another high-level language.
2. Control flow should be easy to understand to help in manual inspection, formal verification and optimization.
3. The translation from Yul to bytecode should be as straightforward as possible.
4. Yul should be suitable for whole-program optimization.

In order to achieve the first and second goal, Yul provides high-level constructs like for loops, if and switch statements and function calls. These should be sufficient for adequately representing the control flow for assembly programs. Therefore, no explicit statements for SWAP, DUP, JUMPDEST, JUMP and JUMPI are provided, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form mul(add(x, y), 7) are preferred over pure opcode statements like 7 y x add mul because in the first form, it is much easier to see which operand is used for which opcode.

Even though it was designed for stack machines, Yul does not expose the complexity of the stack itself. The programmer or auditor should not have to worry about the stack.

The third goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, …) and cleanup of local variables from the stack.

To avoid confusions between concepts like values and references, Yul is statically typed. At the same time, there is a default type (usually the integer word of the target machine) that can always be omitted to help readability.

To keep the language simple and flexible, Yul does not have any built-in operations, functions or types in its pure form. These are added together with their semantics when specifying a dialect of Yul, which allows specializing Yul to the requirements of different target platforms and feature sets.

Currently, there is only one specified dialect of Yul. This dialect uses the EVM opcodes as builtin functions (see below) and defines only the type u256, which is the native 256-bit type of the EVM. Because of that, we will not provide types in the examples below.

**Simple Example**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#simple-example)

The following example program is written in the EVM dialect and computes exponentiation. It can be compiled using solc --strict-assembly. The builtin functions mul and div compute product and division, respectively.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgZnVuY3Rpb24gcG93ZXIoYmFzZSwgZXhwb25lbnQpIC0+IHJlc3VsdAogICAgewogICAgICAgIHN3aXRjaCBleHBvbmVudAogICAgICAgIGNhc2UgMCB7IHJlc3VsdCA6PSAxIH0KICAgICAgICBjYXNlIDEgeyByZXN1bHQgOj0gYmFzZSB9CiAgICAgICAgZGVmYXVsdAogICAgICAgIHsKICAgICAgICAgICAgcmVzdWx0IDo9IHBvd2VyKG11bChiYXNlLCBiYXNlKSwgZGl2KGV4cG9uZW50LCAyKSkKICAgICAgICAgICAgc3dpdGNoIG1vZChleHBvbmVudCwgMikKICAgICAgICAgICAgICAgIGNhc2UgMSB7IHJlc3VsdCA6PSBtdWwoYmFzZSwgcmVzdWx0KSB9CiAgICAgICAgfQogICAgfQp9)

{

**function** power(base, exponent) -> result

{

**switch** exponent

**case** 0 { result := 1 }

**case** 1 { result := base }

**default**

{

result := power(mul(base, base), div(exponent, 2))

**switch** mod(exponent, 2)

**case** 1 { result := mul(base, result) }

}

}

}

It is also possible to implement the same function using a for-loop instead of with recursion. Here, lt(a, b) computes whether a is less than b.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgZnVuY3Rpb24gcG93ZXIoYmFzZSwgZXhwb25lbnQpIC0+IHJlc3VsdAogICAgewogICAgICAgIHJlc3VsdCA6PSAxCiAgICAgICAgZm9yIHsgbGV0IGkgOj0gMCB9IGx0KGksIGV4cG9uZW50KSB7IGkgOj0gYWRkKGksIDEpIH0KICAgICAgICB7CiAgICAgICAgICAgIHJlc3VsdCA6PSBtdWwocmVzdWx0LCBiYXNlKQogICAgICAgIH0KICAgIH0KfQ==)

{

**function** power(base, exponent) -> result

{

result := 1

**for** { **let** i := 0 } lt(i, exponent) { i := add(i, 1) }

{

result := mul(result, base)

}

}

}

At the [end of the section](https://docs.soliditylang.org/en/v0.8.20/yul.html#erc20yul), a complete implementation of the ERC-20 standard can be found.

**Stand-Alone Usage**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#stand-alone-usage)

You can use Yul in its stand-alone form in the EVM dialect using the Solidity compiler. This will use the [Yul object notation](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-object) so that it is possible to refer to code as data to deploy contracts. This Yul mode is available for the commandline compiler (use --strict-assembly) and for the [standard-json interface](https://docs.soliditylang.org/en/v0.8.20/using-the-compiler.html#compiler-api):

{

**"language"**: "Yul",

**"sources"**: { **"input.yul"**: { **"content"**: "{ sstore(0, 1) }" } },

**"settings"**: {

**"outputSelection"**: { **"\*"**: { **"\*"**: ["\*"], **""**: [ "\*" ] } },

**"optimizer"**: { **"enabled"**: **true**, **"details"**: { **"yul"**: **true** } }

}

}

**Warning**

Yul is in active development and bytecode generation is only fully implemented for the EVM dialect of Yul with EVM 1.0 as target.

**Informal Description of Yul**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#informal-description-of-yul)

In the following, we will talk about each individual aspect of the Yul language. In examples, we will use the default EVM dialect.

**Syntax**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#syntax)

Yul parses comments, literals and identifiers in the same way as Solidity, so you can e.g. use // and /\* \*/ to denote comments. There is one exception: Identifiers in Yul can contain dots: ..

Yul can specify “objects” that consist of code, data and sub-objects. Please see [Yul Objects](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-object) below for details on that. In this section, we are only concerned with the code part of such an object. This code part always consists of a curly-braces delimited block. Most tools support specifying just a code block where an object is expected.

Inside a code block, the following elements can be used (see the later sections for more details):

* literals, e.g. 0x123, 42 or "abc" (strings up to 32 characters)
* calls to builtin functions, e.g. add(1, mload(0))
* variable declarations, e.g. let x := 7, let x := add(y, 3) or let x (initial value of 0 is assigned)
* identifiers (variables), e.g. add(3, x)
* assignments, e.g. x := add(y, 3)
* blocks where local variables are scoped inside, e.g. { let x := 3 { let y := add(x, 1) } }
* if statements, e.g. if lt(a, b) { sstore(0, 1) }
* switch statements, e.g. switch mload(0) case 0 { revert() } default { mstore(0, 1) }
* for loops, e.g. for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }
* function definitions, e.g. function f(a, b) -> c { c := add(a, b) }

Multiple syntactical elements can follow each other simply separated by whitespace, i.e. there is no terminating ; or newline required.

**Literals**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#literals)

As literals, you can use:

* Integer constants in decimal or hexadecimal notation.
* ASCII strings (e.g. "abc"), which may contain hex escapes \xNN and Unicode escapes \uNNNN where N are hexadecimal digits.
* Hex strings (e.g. hex"616263").

In the EVM dialect of Yul, literals represent 256-bit words as follows:

* Decimal or hexadecimal constants must be less than 2\*\*256. They represent the 256-bit word with that value as an unsigned integer in big endian encoding.
* An ASCII string is first viewed as a byte sequence, by viewing a non-escape ASCII character as a single byte whose value is the ASCII code, an escape \xNN as single byte with that value, and an escape \uNNNN as the UTF-8 sequence of bytes for that code point. The byte sequence must not exceed 32 bytes. The byte sequence is padded with zeros on the right to reach 32 bytes in length; in other words, the string is stored left-aligned. The padded byte sequence represents a 256-bit word whose most significant 8 bits are the ones from the first byte, i.e. the bytes are interpreted in big endian form.
* A hex string is first viewed as a byte sequence, by viewing each pair of contiguous hex digits as a byte. The byte sequence must not exceed 32 bytes (i.e. 64 hex digits), and is treated as above.

When compiling for the EVM, this will be translated into an appropriate PUSHi instruction. In the following example, 3 and 2 are added resulting in 5 and then the bitwise and with the string “abc” is computed. The final value is assigned to a local variable called x.

The 32-byte limit above does not apply to string literals passed to builtin functions that require literal arguments (e.g. setimmutable or loadimmutable). Those strings never end up in the generated bytecode.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=bGV0IHggOj0gYW5kKCJhYmMiLCBhZGQoMywgMikp)

**let** x := and("abc", add(3, 2))

Unless it is the default type, the type of a literal has to be specified after a colon:

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=Ly8gVGhpcyB3aWxsIG5vdCBjb21waWxlICh1MzIgYW5kIHUyNTYgdHlwZSBub3QgaW1wbGVtZW50ZWQgeWV0KQpsZXQgeCA6PSBhbmQoImFiYyI6dTMyLCBhZGQoMzp1MjU2LCAyOnUyNTYpKQ==)

*// This will not compile (u32 and u256 type not implemented yet)*

**let** x := and("abc":u32, add(3:u256, 2:u256))

**Function Calls**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#function-calls)

Both built-in and user-defined functions (see below) can be called in the same way as shown in the previous example. If the function returns a single value, it can be directly used inside an expression again. If it returns multiple values, they have to be assigned to local variables.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ZnVuY3Rpb24gZih4LCB5KSAtPiBhLCBiIHsgLyogLi4uICovIH0KbXN0b3JlKDB4ODAsIGFkZChtbG9hZCgweDgwKSwgMykpCi8vIEhlcmUsIHRoZSB1c2VyLWRlZmluZWQgZnVuY3Rpb24gYGZgIHJldHVybnMgdHdvIHZhbHVlcy4KbGV0IHgsIHkgOj0gZigxLCBtbG9hZCgwKSk=)

**function** f(x, y) -> a, b { */\* ... \*/* }

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

*// Here, the user-defined function `f` returns two values.*

**let** x, y := f(1, mload(0))

For built-in functions of the EVM, functional expressions can be directly translated to a stream of opcodes: You just read the expression from right to left to obtain the opcodes. In the case of the second line in the example, this is PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE.

For calls to user-defined functions, the arguments are also put on the stack from right to left and this is the order in which argument lists are evaluated. The return values, though, are expected on the stack from left to right, i.e. in this example, y is on top of the stack and x is below it.

**Variable Declarations**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#variable-declarations)

You can use the let keyword to declare variables. A variable is only visible inside the {...}-block it was defined in. When compiling to the EVM, a new stack slot is created that is reserved for the variable and automatically removed again when the end of the block is reached. You can provide an initial value for the variable. If you do not provide a value, the variable will be initialized to zero.

Since variables are stored on the stack, they do not directly influence memory or storage, but they can be used as pointers to memory or storage locations in the built-in functions mstore, mload, sstore and sload. Future dialects might introduce specific types for such pointers.

When a variable is referenced, its current value is copied. For the EVM, this translates to a DUP instruction.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgbGV0IHplcm8gOj0gMAogICAgbGV0IHYgOj0gY2FsbGRhdGFsb2FkKHplcm8pCiAgICB7CiAgICAgICAgbGV0IHkgOj0gYWRkKHNsb2FkKHYpLCAxKQogICAgICAgIHYgOj0geQogICAgfSAvLyB5IGlzICJkZWFsbG9jYXRlZCIgaGVyZQogICAgc3N0b3JlKHYsIHplcm8pCn0gLy8gdiBhbmQgemVybyBhcmUgImRlYWxsb2NhdGVkIiBoZXJl)

{

**let** zero := 0

**let** v := calldataload(zero)

{

**let** y := add(sload(v), 1)

v := y

} *// y is "deallocated" here*

sstore(v, zero)

} *// v and zero are "deallocated" here*

If the declared variable should have a type different from the default type, you denote that following a colon. You can also declare multiple variables in one statement when you assign from a function call that returns multiple values.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=Ly8gVGhpcyB3aWxsIG5vdCBjb21waWxlICh1MzIgYW5kIHUyNTYgdHlwZSBub3QgaW1wbGVtZW50ZWQgeWV0KQp7CiAgICBsZXQgemVybzp1MzIgOj0gMDp1MzIKICAgIGxldCB2OnUyNTYsIHQ6dTMyIDo9IGYoKQogICAgbGV0IHgsIHkgOj0gZygpCn0=)

*// This will not compile (u32 and u256 type not implemented yet)*

{

**let** zero:u32 := 0:u32

**let** v:u256, t:u32 := f()

**let** x, y := g()

}

Depending on the optimiser settings, the compiler can free the stack slots already after the variable has been used for the last time, even though it is still in scope.

**Assignments**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#assignments)

Variables can be assigned to after their definition using the := operator. It is possible to assign multiple variables at the same time. For this, the number and types of the values have to match. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables. The same variable may not occur multiple times on the left-hand side of an assignment, e.g. x, x := f() is invalid.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=bGV0IHYgOj0gMAovLyByZS1hc3NpZ24gdgp2IDo9IDIKbGV0IHQgOj0gYWRkKHYsIDIpCmZ1bmN0aW9uIGYoKSAtPiBhLCBiIHsgfQovLyBhc3NpZ24gbXVsdGlwbGUgdmFsdWVzCnYsIHQgOj0gZigp)

**let** v := 0

*// re-assign v*

v := 2

**let** t := add(v, 2)

**function** f() -> a, b { }

*// assign multiple values*

v, t := f()

**If**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#if)

The if statement can be used for conditionally executing code. No “else” block can be defined. Consider using “switch” instead (see below) if you need multiple alternatives.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=aWYgbHQoY2FsbGRhdGFzaXplKCksIDQpIHsgcmV2ZXJ0KDAsIDApIH0=)

**if** lt(calldatasize(), 4) { revert(0, 0) }

The curly braces for the body are required.

**Switch**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#switch)

You can use a switch statement as an extended version of the if statement. It takes the value of an expression and compares it to several literal constants. The branch corresponding to the matching constant is taken. Contrary to other programming languages, for safety reasons, control flow does not continue from one case to the next. There can be a fallback or default case called default which is taken if none of the literal constants matches.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgbGV0IHggOj0gMAogICAgc3dpdGNoIGNhbGxkYXRhbG9hZCg0KQogICAgY2FzZSAwIHsKICAgICAgICB4IDo9IGNhbGxkYXRhbG9hZCgweDI0KQogICAgfQogICAgZGVmYXVsdCB7CiAgICAgICAgeCA6PSBjYWxsZGF0YWxvYWQoMHg0NCkKICAgIH0KICAgIHNzdG9yZSgwLCBkaXYoeCwgMikpCn0=)

{

**let** x := 0

**switch** calldataload(4)

**case** 0 {

x := calldataload(0x24)

}

**default** {

x := calldataload(0x44)

}

sstore(0, div(x, 2))

}

The list of cases is not enclosed by curly braces, but the body of a case does require them.

**Loops**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#loops)

Yul supports for-loops which consist of a header containing an initializing part, a condition, a post-iteration part and a body. The condition has to be an expression, while the other three are blocks. If the initializing part declares any variables at the top level, the scope of these variables extends to all other parts of the loop.

The break and continue statements can be used in the body to exit the loop or skip to the post-part, respectively.

The following example computes the sum of an area in memory.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgbGV0IHggOj0gMAogICAgZm9yIHsgbGV0IGkgOj0gMCB9IGx0KGksIDB4MTAwKSB7IGkgOj0gYWRkKGksIDB4MjApIH0gewogICAgICAgIHggOj0gYWRkKHgsIG1sb2FkKGkpKQogICAgfQp9)

{

**let** x := 0

**for** { **let** i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {

x := add(x, mload(i))

}

}

For loops can also be used as a replacement for while loops: Simply leave the initialization and post-iteration parts empty.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgbGV0IHggOj0gMAogICAgbGV0IGkgOj0gMAogICAgZm9yIHsgfSBsdChpLCAweDEwMCkgeyB9IHsgICAgIC8vIHdoaWxlKGkgPCAweDEwMCkKICAgICAgICB4IDo9IGFkZCh4LCBtbG9hZChpKSkKICAgICAgICBpIDo9IGFkZChpLCAweDIwKQogICAgfQp9)

{

**let** x := 0

**let** i := 0

**for** { } lt(i, 0x100) { } { *// while(i < 0x100)*

x := add(x, mload(i))

i := add(i, 0x20)

}

}

**Function Declarations**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#function-declarations)

Yul allows the definition of functions. These should not be confused with functions in Solidity since they are never part of an external interface of a contract and are part of a namespace separate from the one for Solidity functions.

For the EVM, Yul functions take their arguments (and a return PC) from the stack and also put the results onto the stack. User-defined functions and built-in functions are called in exactly the same way.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function.

Functions declare parameters and return variables, similar to Solidity. To return a value, you assign it to the return variable(s).

If you call a function that returns multiple values, you have to assign them to multiple variables using a, b := f(x) or let a, b := f(x).

The leave statement can be used to exit the current function. It works like the return statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

Note that the EVM dialect has a built-in function called return that quits the full execution context (internal message call) and not just the current yul function.

The following example implements the power function by square-and-multiply.

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=ewogICAgZnVuY3Rpb24gcG93ZXIoYmFzZSwgZXhwb25lbnQpIC0+IHJlc3VsdCB7CiAgICAgICAgc3dpdGNoIGV4cG9uZW50CiAgICAgICAgY2FzZSAwIHsgcmVzdWx0IDo9IDEgfQogICAgICAgIGNhc2UgMSB7IHJlc3VsdCA6PSBiYXNlIH0KICAgICAgICBkZWZhdWx0IHsKICAgICAgICAgICAgcmVzdWx0IDo9IHBvd2VyKG11bChiYXNlLCBiYXNlKSwgZGl2KGV4cG9uZW50LCAyKSkKICAgICAgICAgICAgc3dpdGNoIG1vZChleHBvbmVudCwgMikKICAgICAgICAgICAgICAgIGNhc2UgMSB7IHJlc3VsdCA6PSBtdWwoYmFzZSwgcmVzdWx0KSB9CiAgICAgICAgfQogICAgfQp9)

{

**function** power(base, exponent) -> result {

**switch** exponent

**case** 0 { result := 1 }

**case** 1 { result := base }

**default** {

result := power(mul(base, base), div(exponent, 2))

**switch** mod(exponent, 2)

**case** 1 { result := mul(base, result) }

}

}

}

**Specification of Yul**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#specification-of-yul)

This chapter describes Yul code formally. Yul code is usually placed inside Yul objects, which are explained in their own chapter.

Block = '{' Statement\* '}'

Statement =

Block |

FunctionDefinition |

VariableDeclaration |

Assignment |

If |

Expression |

Switch |

ForLoop |

BreakContinue |

Leave

FunctionDefinition =

'function' Identifier '(' TypedIdentifierList? ')'

( '->' TypedIdentifierList )? Block

VariableDeclaration =

'let' TypedIdentifierList ( ':=' Expression )?

Assignment =

IdentifierList ':=' Expression

Expression =

FunctionCall | Identifier | Literal

If =

'if' Expression Block

Switch =

'switch' Expression ( Case+ Default? | Default )

Case =

'case' Literal Block

Default =

'default' Block

ForLoop =

'for' Block Expression Block Block

BreakContinue =

'break' | 'continue'

Leave = 'leave'

FunctionCall =

Identifier '(' ( Expression ( ',' Expression )\* )? ')'

Identifier = [a-zA-Z\_$] [a-zA-Z\_$0-9.]\*

IdentifierList = Identifier ( ',' Identifier)\*

TypeName = Identifier

TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )\*

Literal =

(NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?

NumberLiteral = HexNumber | DecimalNumber

StringLiteral = '"' ([^"\r\n\\] | '\\' .)\* '"'

TrueLiteral = 'true'

FalseLiteral = 'false'

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

DecimalNumber = [0-9]+

**Restrictions on the Grammar**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#restrictions-on-the-grammar)

Apart from those directly imposed by the grammar, the following restrictions apply:

Switches must have at least one case (including the default case). All case values need to have the same type and distinct values. If all possible values of the expression type are covered, a default case is not allowed (i.e. a switch with a bool expression that has both a true and a false case do not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return variables of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed. The same variable name cannot occur more than once in the left-hand-side of an assignment or variable declaration.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

A continue or break statement can only be used inside the body of a for-loop, as follows. Consider the innermost loop that contains the statement. The loop and the statement must be in the same function, or both must be at the top level. The statement must be in the loop’s body block; it cannot be in the loop’s initialization block or update block. It is worth emphasizing that this restriction applies just to the innermost loop that contains the continue or break statement: this innermost loop, and therefore the continue or break statement, may appear anywhere in an outer loop, possibly in an outer loop’s initialization block or update block. For example, the following is legal, because the break occurs in the body block of the inner loop, despite also occurring in the update block of the outer loop:

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=Zm9yIHt9IHRydWUgeyBmb3Ige30gdHJ1ZSB7fSB7IGJyZWFrIH0gfQp7Cn0=)

**for** {} true { **for** {} true {} { break } }

{

}

The condition part of the for-loop has to evaluate to exactly one value.

The leave statement can only be used inside a function.

Functions cannot be defined anywhere inside for loop init blocks.

Literals cannot be larger than their type. The largest type defined is 256-bit wide.

During assignments and function calls, the types of the respective values have to match. There is no implicit type conversion. Type conversion in general can only be achieved if the dialect provides an appropriate built-in function that takes a value of one type and returns a value of a different type.

**Scoping Rules**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#scoping-rules)

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (FunctionDefinition, VariableDeclaration) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks): Functions are visible in the whole block (even before their definitions) while variables are only visible starting from the statement after the VariableDeclaration.

In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

As an exception to the general scoping rule, the scope of the “init” part of the for-loop (the first block) extends across all other parts of the for loop. This means that variables (and functions) declared in the init part (but not inside a block inside the init part) are visible in all other parts of the for-loop.

Identifiers declared in the other parts of the for loop respect the regular syntactical scoping rules.

This means a for-loop of the form for { I... } C { P... } { B... } is equivalent to { I... for {} C { P... } { B... } }.

The parameters and return parameters of functions are visible in the function body and their names have to be distinct.

Inside functions, it is not possible to reference a variable that was declared outside of that function.

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not possible to reference it because it was declared outside the current function.

**Formal Specification**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#formal-specification)

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. As builtin functions can have side effects, E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM).

If the AST node is a statement, E returns the two state objects and a “mode”, which is used for the break, continue and leave statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state L is a mapping of identifiers i to values v, denoted as L[i] = v.

For an identifier v, let $v be the name of the identifier.

We will use a destructuring notation for the AST nodes.

E(G, L, <{St1, ..., Stn}>: Block) =

let G1, L1, mode = E(G, L, St1, ..., Stn)

let L2 be a restriction of L1 to the identifiers of L

G1, L2, mode

E(G, L, St1, ..., Stn: Statement) =

if n is zero:

G, L, regular

else:

let G1, L1, mode = E(G, L, St1)

if mode is regular then

E(G1, L1, St2, ..., Stn)

otherwise

G1, L1, mode

E(G, L, FunctionDefinition) =

G, L, regular

E(G, L, <let var\_1, ..., var\_n := rhs>: VariableDeclaration) =

E(G, L, <var\_1, ..., var\_n := rhs>: Assignment)

E(G, L, <let var\_1, ..., var\_n>: VariableDeclaration) =

let L1 be a copy of L where L1[$var\_i] = 0 for i = 1, ..., n

G, L1, regular

E(G, L, <var\_1, ..., var\_n := rhs>: Assignment) =

let G1, L1, v1, ..., vn = E(G, L, rhs)

let L2 be a copy of L1 where L2[$var\_i] = vi for i = 1, ..., n

G1, L2, regular

E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =

if n >= 1:

let G1, L1, mode = E(G, L, i1, ..., in)

// mode has to be regular or leave due to the syntactic restrictions

if mode is leave then

G1, L1 restricted to variables of L, leave

otherwise

let G2, L2, mode = E(G1, L1, for {} condition post body)

G2, L2 restricted to variables of L, mode

else:

let G1, L1, v = E(G, L, condition)

if v is false:

G1, L1, regular

else:

let G2, L2, mode = E(G1, L, body)

if mode is break:

G2, L2, regular

otherwise if mode is leave:

G2, L2, leave

else:

G3, L3, mode = E(G2, L2, post)

if mode is leave:

G3, L3, leave

otherwise

E(G3, L3, for {} condition post body)

E(G, L, break: BreakContinue) =

G, L, break

E(G, L, continue: BreakContinue) =

G, L, continue

E(G, L, leave: Leave) =

G, L, leave

E(G, L, <if condition body>: If) =

let G0, L0, v = E(G, L, condition)

if v is true:

E(G0, L0, body)

else:

G0, L0, regular

E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =

E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})

E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =

let G0, L0, v = E(G, L, condition)

// i = 1 .. n

// Evaluate literals, context doesn't matter

let \_, \_, v1 = E(G0, L0, l1)

...

let \_, \_, vn = E(G0, L0, ln)

if there exists smallest i such that vi = v:

E(G0, L0, sti)

else:

E(G0, L0, st')

E(G, L, <name>: Identifier) =

G, L, L[$name]

E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =

G1, L1, vn = E(G, L, argn)

...

G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)

Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)

Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>

be the function of name $fname visible at the point of the call.

Let L' be a new local state such that

L'[$parami] = vi and L'[$reti] = 0 for all i.

Let G'', L'', mode = E(Gn, L', block)

G'', Ln, L''[$ret1], ..., L''[$retm]

E(G, L, l: StringLiteral) = G, L, str(l),

where str is the string evaluation function,

which for the EVM dialect is defined in the section 'Literals' above

E(G, L, n: HexNumber) = G, L, hex(n)

where hex is the hexadecimal evaluation function,

which turns a sequence of hexadecimal digits into their big endian value

E(G, L, n: DecimalNumber) = G, L, dec(n),

where dec is the decimal evaluation function,

which turns a sequence of decimal digits into their big endian value

**EVM Dialect**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#evm-dialect)

The default dialect of Yul currently is the EVM dialect for the currently selected version of the EVM. with a version of the EVM. The only type available in this dialect is u256, the 256-bit native type of the Ethereum Virtual Machine. Since it is the default type of this dialect, it can be omitted.

The following table lists all builtin functions (depending on the EVM version) and provides a short description of the semantics of the function / opcode. This document does not want to be a full description of the Ethereum virtual machine. Please refer to a different document if you are interested in the precise semantics.

Opcodes marked with - do not return a result and all others return exactly one value. Opcodes marked with F, H, B, C, I, L and P are present since Frontier, Homestead, Byzantium, Constantinople, Istanbul, London or Paris respectively.

In the following, mem[a...b) signifies the bytes of memory starting at position a up to but not including position b and storage[p] signifies the storage contents at slot p.

Since Yul manages local variables and control-flow, opcodes that interfere with these features are not available. This includes the dup and swap instructions as well as jump instructions, labels and the push instructions.

| **Instruction** |  |  | **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 or 0 if y == 0 |
| sdiv(x, y) |  | F | x / y, for signed numbers in two’s complement, 0 if y == 0 |
| mod(x, y) |  | F | x % y, 0 if y == 0 |
| smod(x, y) |  | F | x % y, for signed numbers in two’s complement, 0 if y == 0 |
| exp(x, y) |  | F | x to the power of y |
| not(x) |  | F | bitwise “not” of 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 |
| 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 | signed arithmetic shift right y by x bits |
| addmod(x, y, m) |  | F | (x + y) % m with arbitrary precision arithmetic, 0 if m == 0 |
| mulmod(x, y, m) |  | F | (x \* y) % m with arbitrary precision arithmetic, 0 if m == 0 |
| 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))) |
| pc() |  | F | current position in code |
| pop(x) | *-* | F | discard value x |
| 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 |
| 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 |
| selfbalance() |  | I | equivalent to balance(address()), but cheaper |
| 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 |
| 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; returns 0 on error |
| 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 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 |
| 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 [See more](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-call-return-area) |
| 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 [See more](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-call-return-area) |
| delegatecall(g, a, in, insize, out, outsize) |  | H | identical to callcode but also keep caller and callvalue [See more](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-call-return-area) |
| staticcall(g, a, in, insize, out, outsize) |  | B | identical to call(g, a, 0, in, insize, out, outsize) but do not allow state modifications [See more](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-call-return-area) |
| 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)) |
| selfdestruct(a) | *-* | F | end execution, destroy current contract and send funds to a (deprecated) |
| invalid() | *-* | F | end execution with invalid instruction |
| log0(p, s) | *-* | F | log data mem[p…(p+s)) |
| log1(p, s, t1) | *-* | F | log data mem[p…(p+s)) with topic t1 |
| log2(p, s, t1, t2) | *-* | F | log data mem[p…(p+s)) with topics t1, t2 |
| log3(p, s, t1, t2, t3) | *-* | F | log data mem[p…(p+s)) with topics t1, t2, t3 |
| log4(p, s, t1, t2, t3, t4) | *-* | F | log data mem[p…(p+s)) with topics t1, t2, t3, t4 |
| chainid() |  | I | ID of the executing chain (EIP-1344) |
| basefee() |  | L | current block’s base fee (EIP-3198 and EIP-1559) |
| 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 (see note below) |
| prevrandao() |  | P | randomness provided by the beacon chain (see note below) |
| gaslimit() |  | F | block gas limit of the current block |

**Note**

The call\* instructions use the out and outsize parameters to define an area in memory where the return or failure data is placed. This area is written to depending on how many bytes the called contract returns. If it returns more data, only the first outsize bytes are written. You can access the rest of the data using the returndatacopy opcode. If it returns less data, then the remaining bytes are not touched at all. You need to use the returndatasize opcode to check which part of this memory area contains the return data. The remaining bytes will retain their values as of before the call.

**Note**

The difficulty() instruction is disallowed in EVM version >= Paris. With the Paris network upgrade the semantics of the instruction that was previously called difficulty have been changed and the instruction was renamed to prevrandao. It can now return arbitrary values in the full 256-bit range, whereas the highest recorded difficulty value within Ethash was ~54 bits. This change is described in [EIP-4399](https://eips.ethereum.org/EIPS/eip-4399). Please note that irrelevant to which EVM version is selected in the compiler, the semantics of instructions depend on the final chain of deployment.

**Warning**

From version 0.8.18 and up, the use of selfdestruct in both Solidity and Yul will trigger a deprecation warning, since the SELFDESTRUCT opcode will eventually undergo breaking changes in behaviour as stated in [EIP-6049](https://eips.ethereum.org/EIPS/eip-6049).

In some internal dialects, there are additional functions:

**datasize, dataoffset, datacopy[ℑ](https://docs.soliditylang.org/en/v0.8.20/yul.html" \l "datasize-dataoffset-datacopy" \o "Permalink to this heading)**

The functions datasize(x), dataoffset(x) and datacopy(t, f, l) are used to access other parts of a Yul object.

datasize and dataoffset can only take string literals (the names of other objects) as arguments and return the size and offset in the data area, respectively. For the EVM, the datacopy function is equivalent to codecopy.

**setimmutable, loadimmutable[ℑ](https://docs.soliditylang.org/en/v0.8.20/yul.html" \l "setimmutable-loadimmutable" \o "Permalink to this heading)**

The functions setimmutable(offset, "name", value) and loadimmutable("name") are used for the immutable mechanism in Solidity and do not nicely map to pure Yul. The call to setimmutable(offset, "name", value) assumes that the runtime code of the contract containing the given named immutable was copied to memory at offset offset and will write value to all positions in memory (relative to offset) that contain the placeholder that was generated for calls to loadimmutable("name") in the runtime code.

**linkersymbol[ℑ](https://docs.soliditylang.org/en/v0.8.20/yul.html" \l "linkersymbol" \o "Permalink to this heading)**

The function linkersymbol("library\_id") is a placeholder for an address literal to be substituted by the linker. Its first and only argument must be a string literal and uniquely represents the address to be inserted. Identifiers can be arbitrary but when the compiler produces Yul code from Solidity sources, it uses a library name qualified with the name of the source unit that defines that library. To link the code with a particular library address, the same identifier must be provided to the --libraries option on the command line.

For example this code

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=bGV0IGEgOj0gbGlua2Vyc3ltYm9sKCJmaWxlLnNvbDpNYXRoIik=)

**let** a := linkersymbol("file.sol:Math")

is equivalent to

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=bGV0IGEgOj0gMHgxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkw)

**let** a := 0x1234567890123456789012345678901234567890

when the linker is invoked with --libraries "file.sol:Math=0x1234567890123456789012345678901234567890 option.

See [Using the Commandline Compiler](https://docs.soliditylang.org/en/v0.8.20/using-the-compiler.html#commandline-compiler) for details about the Solidity linker.

**memoryguard[ℑ](https://docs.soliditylang.org/en/v0.8.20/yul.html" \l "memoryguard" \o "Permalink to this heading)**

This function is available in the EVM dialect with objects. The caller of let ptr := memoryguard(size) (where size has to be a literal number) promises that they only use memory in either the range [0, size) or the unbounded range starting at ptr.

Since the presence of a memoryguard call indicates that all memory access adheres to this restriction, it allows the optimizer to perform additional optimization steps, for example the stack limit evader, which attempts to move stack variables that would otherwise be unreachable to memory.

The Yul optimizer promises to only use the memory range [size, ptr) for its purposes. If the optimizer does not need to reserve any memory, it holds that ptr == size.

memoryguard can be called multiple times, but needs to have the same literal as argument within one Yul subobject. If at least one memoryguard call is found in a subobject, the additional optimiser steps will be run on it.

**verbatim**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#verbatim)

The set of verbatim... builtin functions lets you create bytecode for opcodes that are not known to the Yul compiler. It also allows you to create bytecode sequences that will not be modified by the optimizer.

The functions are verbatim\_<n>i\_<m>o("<data>", ...), where

* n is a decimal between 0 and 99 that specifies the number of input stack slots / variables
* m is a decimal between 0 and 99 that specifies the number of output stack slots / variables
* data is a string literal that contains the sequence of bytes

If you for example want to define a function that multiplies the input by two, without the optimizer touching the constant two, you can use

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=bGV0IHggOj0gY2FsbGRhdGFsb2FkKDApCmxldCBkb3VibGUgOj0gdmVyYmF0aW1fMWlfMW8oaGV4IjYwMDIwMiIsIHgp)

**let** x := calldataload(0)

**let** double := verbatim\_1i\_1o(hex"600202", x)

This code will result in a dup1 opcode to retrieve x (the optimizer might directly re-use result of the calldataload opcode, though) directly followed by 600202. The code is assumed to consume the copied value of x and produce the result on the top of the stack. The compiler then generates code to allocate a stack slot for double and store the result there.

As with all opcodes, the arguments are arranged on the stack with the leftmost argument on the top, while the return values are assumed to be laid out such that the rightmost variable is at the top of the stack.

Since verbatim can be used to generate arbitrary opcodes or even opcodes unknown to the Solidity compiler, care has to be taken when using verbatim together with the optimizer. Even when the optimizer is switched off, the code generator has to determine the stack layout, which means that e.g. using verbatim to modify the stack height can lead to undefined behaviour.

The following is a non-exhaustive list of restrictions on verbatim bytecode that are not checked by the compiler. Violations of these restrictions can result in undefined behaviour.

* Control-flow should not jump into or out of verbatim blocks, but it can jump within the same verbatim block.
* Stack contents apart from the input and output parameters should not be accessed.
* The stack height difference should be exactly m - n (output slots minus input slots).
* Verbatim bytecode cannot make any assumptions about the surrounding bytecode. All required parameters have to be passed in as stack variables.

The optimizer does not analyze verbatim bytecode and always assumes that it modifies all aspects of state and thus can only do very few optimizations across verbatim function calls.

The optimizer treats verbatim bytecode as an opaque block of code. It will not split it but might move, duplicate or combine it with identical verbatim bytecode blocks. If a verbatim bytecode block is unreachable by the control-flow, it can be removed.

**Warning**

During discussions about whether or not EVM improvements might break existing smart contracts, features inside verbatim cannot receive the same consideration as those used by the Solidity compiler itself.

**Note**

To avoid confusion, all identifiers starting with the string verbatim are reserved and cannot be used for user-defined identifiers.

**Specification of Yul Object**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#specification-of-yul-object)

Yul objects are used to group named code and data sections. The functions datasize, dataoffset and datacopy can be used to access these sections from within code. Hex strings can be used to specify data in hex encoding, regular strings in native encoding. For code, datacopy will access its assembled binary representation.

Object = 'object' StringLiteral '{' Code ( Object | Data )\* '}'

Code = 'code' Block

Data = 'data' StringLiteral ( HexLiteral | StringLiteral )

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

StringLiteral = '"' ([^"\r\n\\] | '\\' .)\* '"'

Above, Block refers to Block in the Yul code grammar explained in the previous chapter.

**Note**

An object with a name that ends in \_deployed is treated as deployed code by the Yul optimizer. The only consequence of this is a different gas cost heuristic in the optimizer.

**Note**

Data objects or sub-objects whose names contain a . can be defined but it is not possible to access them through datasize, dataoffset or datacopy because . is used as a separator to access objects inside another object.

**Note**

The data object called ".metadata" has a special meaning: It cannot be accessed from code and is always appended to the very end of the bytecode, regardless of its position in the object.

Other data objects with special significance might be added in the future, but their names will always start with a ..

An example Yul Object is shown below:

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=)

*// A contract consists of a single object with sub-objects representing*

*// the code to be deployed or other contracts it can create.*

*// The single "code" node is the executable code of the object.*

*// Every (other) named object or data section is serialized and*

*// made accessible to the special built-in functions datacopy / dataoffset / datasize*

*// The current object, sub-objects and data items inside the current object*

*// are in scope.*

**object** "Contract1" {

*// This is the constructor code of the contract.*

**code** {

**function** allocate(size) -> ptr {

ptr := mload(0x40)

*// Note that Solidity generated IR code reserves memory offset ``0x60`` as well, but a pure Yul object is free to use memory as it chooses.*

**if** iszero(ptr) { ptr := 0x60 }

mstore(0x40, add(ptr, size))

}

*// first create "Contract2"*

**let** size := **datasize**("Contract2")

**let** offset := allocate(size)

*// This will turn into codecopy for EVM*

**datacopy**(offset, **dataoffset**("Contract2"), size)

*// constructor parameter is a single number 0x1234*

mstore(add(offset, size), 0x1234)

pop(create(0, offset, add(size, 32)))

*// now return the runtime object (the currently*

*// executing code is the constructor code)*

size := **datasize**("Contract1\_deployed")

offset := allocate(size)

*// This will turn into a memory->memory copy for Ewasm and*

*// a codecopy for EVM*

**datacopy**(offset, **dataoffset**("Contract1\_deployed"), size)

return(offset, size)

}

**data** "Table2" hex"4123"

**object** "Contract1\_deployed" {

**code** {

**function** allocate(size) -> ptr {

ptr := mload(0x40)

*// Note that Solidity generated IR code reserves memory offset ``0x60`` as well, but a pure Yul object is free to use memory as it chooses.*

**if** iszero(ptr) { ptr := 0x60 }

mstore(0x40, add(ptr, size))

}

*// runtime code*

mstore(0, "Hello, World!")

return(0, 0x20)

}

}

*// Embedded object. Use case is that the outside is a factory contract,*

*// and Contract2 is the code to be created by the factory*

**object** "Contract2" {

**code** {

*// code here ...*

}

**object** "Contract2\_deployed" {

**code** {

*// code here ...*

}

}

**data** "Table1" hex"4123"

}

}

**Yul Optimizer**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#yul-optimizer)

The Yul optimizer operates on Yul code and uses the same language for input, output and intermediate states. This allows for easy debugging and verification of the optimizer.

Please refer to the general [optimizer documentation](https://docs.soliditylang.org/en/v0.8.20/internals/optimizer.html#optimizer) for more details about the different optimization stages and how to use the optimizer.

If you want to use Solidity in stand-alone Yul mode, you activate the optimizer using --optimize and optionally specify the [expected number of contract executions](https://docs.soliditylang.org/en/v0.8.20/internals/optimizer.html#optimizer-parameter-runs) with --optimize-runs:

solc --strict-assembly --optimize --optimize-runs 200

In Solidity mode, the Yul optimizer is activated together with the regular optimizer.

**Optimization Step Sequence**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#optimization-step-sequence)

Detailed information regrading the optimization sequence as well a list of abbreviations is available in the [optimizer docs](https://docs.soliditylang.org/en/v0.8.20/internals/optimizer.html#optimizer-steps).

**Complete ERC20 Example**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/yul.html#complete-erc20-example)

[open in Remix](https://remix.ethereum.org/?#language=yul&version=0.8.20&code=)

**object** "Token" {

**code** {

*// Store the creator in slot zero.*

sstore(0, caller())

*// Deploy the contract*

**datacopy**(0, **dataoffset**("runtime"), **datasize**("runtime"))

return(0, **datasize**("runtime"))

}

**object** "runtime" {

**code** {

*// Protection against sending Ether*

require(iszero(callvalue()))

*// Dispatcher*

**switch** selector()

**case** 0x70a08231 */\* "balanceOf(address)" \*/* {

returnUint(balanceOf(decodeAsAddress(0)))

}

**case** 0x18160ddd */\* "totalSupply()" \*/* {

returnUint(totalSupply())

}

**case** 0xa9059cbb */\* "transfer(address,uint256)" \*/* {

transfer(decodeAsAddress(0), decodeAsUint(1))

returnTrue()

}

**case** 0x23b872dd */\* "transferFrom(address,address,uint256)" \*/* {

transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))

returnTrue()

}

**case** 0x095ea7b3 */\* "approve(address,uint256)" \*/* {

approve(decodeAsAddress(0), decodeAsUint(1))

returnTrue()

}

**case** 0xdd62ed3e */\* "allowance(address,address)" \*/* {

returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))

}

**case** 0x40c10f19 */\* "mint(address,uint256)" \*/* {

mint(decodeAsAddress(0), decodeAsUint(1))

returnTrue()

}

**default** {

revert(0, 0)

}

**function** mint(account, amount) {

require(calledByOwner())

mintTokens(amount)

addToBalance(account, amount)

emitTransfer(0, account, amount)

}

**function** transfer(to, amount) {

executeTransfer(caller(), to, amount)

}

**function** approve(spender, amount) {

revertIfZeroAddress(spender)

setAllowance(caller(), spender, amount)

emitApproval(caller(), spender, amount)

}

**function** transferFrom(from, to, amount) {

decreaseAllowanceBy(from, caller(), amount)

executeTransfer(from, to, amount)

}

**function** executeTransfer(from, to, amount) {

revertIfZeroAddress(to)

deductFromBalance(from, amount)

addToBalance(to, amount)

emitTransfer(from, to, amount)

}

*/\* ---------- calldata decoding functions ----------- \*/*

**function** selector() -> s {

s := div(calldataload(0), 0x100000000000000000000000000000000000000000000000000000000)

}

**function** decodeAsAddress(offset) -> v {

v := decodeAsUint(offset)

**if** iszero(iszero(and(v, not(0xffffffffffffffffffffffffffffffffffffffff)))) {

revert(0, 0)

}

}

**function** decodeAsUint(offset) -> v {

**let** pos := add(4, mul(offset, 0x20))

**if** lt(calldatasize(), add(pos, 0x20)) {

revert(0, 0)

}

v := calldataload(pos)

}

*/\* ---------- calldata encoding functions ---------- \*/*

**function** returnUint(v) {

mstore(0, v)

return(0, 0x20)

}

**function** returnTrue() {

returnUint(1)

}

*/\* -------- events ---------- \*/*

**function** emitTransfer(from, to, amount) {

**let** signatureHash := 0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef

emitEvent(signatureHash, from, to, amount)

}

**function** emitApproval(from, spender, amount) {

**let** signatureHash := 0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925

emitEvent(signatureHash, from, spender, amount)

}

**function** emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {

mstore(0, nonIndexed)

log3(0, 0x20, signatureHash, indexed1, indexed2)

}

*/\* -------- storage layout ---------- \*/*

**function** ownerPos() -> p { p := 0 }

**function** totalSupplyPos() -> p { p := 1 }

**function** accountToStorageOffset(account) -> offset {

offset := add(0x1000, account)

}

**function** allowanceStorageOffset(account, spender) -> offset {

offset := accountToStorageOffset(account)

mstore(0, offset)

mstore(0x20, spender)

offset := keccak256(0, 0x40)

}

*/\* -------- storage access ---------- \*/*

**function** owner() -> o {

o := sload(ownerPos())

}

**function** totalSupply() -> supply {

supply := sload(totalSupplyPos())

}

**function** mintTokens(amount) {

sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))

}

**function** balanceOf(account) -> bal {

bal := sload(accountToStorageOffset(account))

}

**function** addToBalance(account, amount) {

**let** offset := accountToStorageOffset(account)

sstore(offset, safeAdd(sload(offset), amount))

}

**function** deductFromBalance(account, amount) {

**let** offset := accountToStorageOffset(account)

**let** bal := sload(offset)

require(lte(amount, bal))

sstore(offset, sub(bal, amount))

}

**function** allowance(account, spender) -> amount {

amount := sload(allowanceStorageOffset(account, spender))

}

**function** setAllowance(account, spender, amount) {

sstore(allowanceStorageOffset(account, spender), amount)

}

**function** decreaseAllowanceBy(account, spender, amount) {

**let** offset := allowanceStorageOffset(account, spender)

**let** currentAllowance := sload(offset)

require(lte(amount, currentAllowance))

sstore(offset, sub(currentAllowance, amount))

}

*/\* ---------- utility functions ---------- \*/*

**function** lte(a, b) -> r {

r := iszero(gt(a, b))

}

**function** gte(a, b) -> r {

r := iszero(lt(a, b))

}

**function** safeAdd(a, b) -> r {

r := add(a, b)

**if** or(lt(r, a), lt(r, b)) { revert(0, 0) }

}

**function** calledByOwner() -> cbo {

cbo := eq(owner(), caller())

}

**function** revertIfZeroAddress(addr) {

require(addr)

}

**function** require(condition) {

**if** iszero(condition) { revert(0, 0) }

}

}

}

}