# Lesson 6

## Yul continued

## Accessing Variables

For dynamic calldata arrays, you can access their calldata offset (in bytes) and length (number of elements) using `x.offset` and `x.length`.
Both expressions can also be assigned to, but as for the static case, no validation will be performed to ensure that the resulting data area is within the bounds of `calldatasize()`.

For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, their "address" is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x.slot`, and to retrieve the byte-offset you use `x.offset`. Using `x` itself will result in an error.
You can also assign to the `.slot` part of a local storage variable pointer. For these (structs, arrays or mappings), the `.offset` part is always zero.
It is not possible to assign to the `.slot` or `.offset` part of a state variable, though.

For example

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b.slot))
        }
    }
}
```

We can access addresses and function selectors associated with a function with `fun.selector` and `fun.address` for function fun

For example

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Assigns a new selector and address to the return variable @fun
    function combineToFunctionPointer(address newAddress, uint newSelector)
public pure returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

## Memoryguard keyword

```
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

## Verbatim keyword

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

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

## What is missing in Yul ?

Yul tries to hide some of the complexity, therefore the following opcodes are not provided
`SWAP`, `DUP`, `JUMPDEST`, `JUMP` and `JUMPI`, `PUSH`
The `POP` instruction just discards the value
See the table in https://docs.soliditylang.org/en/v0.8.15/yul.html#evm-dialect

# Yul in stand alone mode

Yul can be used in stand alone mode

As an example

```
// 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)
            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(offset, add(size, 32), 0))

        // 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)
                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 ERC20 Example

See Documentation

# Yul+

See introduction from Fuel labs.

Yul+ adds

- Memory structures (mstruct)
- Enums (enum)
- Constants (const)
- Ethereum standard ABI signature/topic generation (sig"function ...", topic"event ...)
- Booleans (true, false)
- Safe math (over/under flow protection for addition, subtraction, multiplication)
- Injected methods (mslice and require)

# IDE Support

## Yul in Remix

Yul is supported (but currently not Yul+)

### Yul+ support in Foundry

Yul template

From Matt Solomon

1. Build with `forge build --extra-output ir` OR add `extra-output = ["ir"]` to your config
2. Run `cat ./out/<file>.sol/<contract>.json | jq -r .ir | perl -pe 's/\\n/\n/g' > ir.sol` (file is .sol for syntax highlighting)

## Viewing Yul output from the solidity compiler

Use the --ir flag with the solidity compiler to get a Yul version, it can be useful to clarify what is happening in your solidity code.

# Huff

"Huff enables the construction of EVM assembly macros - blocks of bytecode that can be rigorously tested and evaluated. Macros can themselves be composed of Huff macros.

Huff doesn't hide the workings of the EVM behind syntactic sugar. In fact, Huff doesn't hide anything at all. Huff does not have variables, instead directly exposing the EVM's program stack to the developer to be directly manipulated."

Huff supports tables of jump destinations integrated directly into the contract bytecode. This is to enable efficient program execution flows by using jump tables instead of conditional branching.

There are only two fundamental building blocks to a Huff program:

- Macros
- Jump tables (and packed jump tables)

Example Macros

```
template <p1,p2>
#define macro POINT_DOUBLE = takes(3) returns(3) {
    <p1> dup3 callvalue shl
    swap3 dup4 mulmod
    <p2> dup2 callvalue shl
    dup2 dup1 dup1 dup4 dup10
    mulmod dup2 sub swap8
    dup1 mulmod 0x03 mul
    dup2 dup2 dup1
    mulmod dup9 callvalue shl add swap8
    dup9 add mulmod swap3 mulmod add swap2
    <p2> swap2 mulmod <p1> sub
}
```

# SLOAD2 and SSTORE2

Repo

SLOAD2 is a set of Solidity libraries for writing and reading contract storage paying a fraction of the cost, it uses contract code as storage, writing data takes the form of contract creations and reading data uses EXTCODECOPY.

SSTORE2

Repo