An Operational Semantics for Yul

Yu-Yang Lin Trinity College Dublin
Vasileios Koutavas Trinity College Dublin & Lero
Nikos Tzevelekos Queen Mary University of London

Research supported in part by Science Foundation Ireland grant 13/RC/2094_2 (Lero), Cisco/Silicon Valley Community Fund, and Ethereum Foundation grant FY23-1127

Background: Ethereum

Popular decentralized blockchain with **smart contract** functionality

- June 2024: Total Value Locked of USD 59.937 billion

EVM: Ethereum Virtual machine

- Environment for executing smart contracts on the Ethereum blockchain
- Stack machine like the JVM

Smart Contracts:

- Immutable scripts stored on a blockchain
- Executed within transactions
- On Ethereum, most are written in Solidity and compiled to EVM bytecode

Ethereum and High-Valued Hacks

Prime target for security incidents:

- USD 686m lost across 224 security incidents in 2023 [CertiK Hack3d: Web3 Report]
- USD 387.8m stolen across 86 incidents in 2024 Q3 (55.5% of all incidents) [CertiK 2024 Q3 Report]
- 71 DeFi exploits between 2020 and 2023 [chainsec.io/defi-hacks]

Some Notable Hacks:

- **DAO hack (2016):** Reentrancy attack, 3.6m ETH (USD ~60m then, USD 8.8b today)
- **Parity Wallet Hacks (2017)**: 666k ETH (USD ~184m then, USD 1.6b today)
- **Poly Network Hack (2021):** USD ~611 million then
- **Ronin Network Hack (2022):** USD ~614 million then

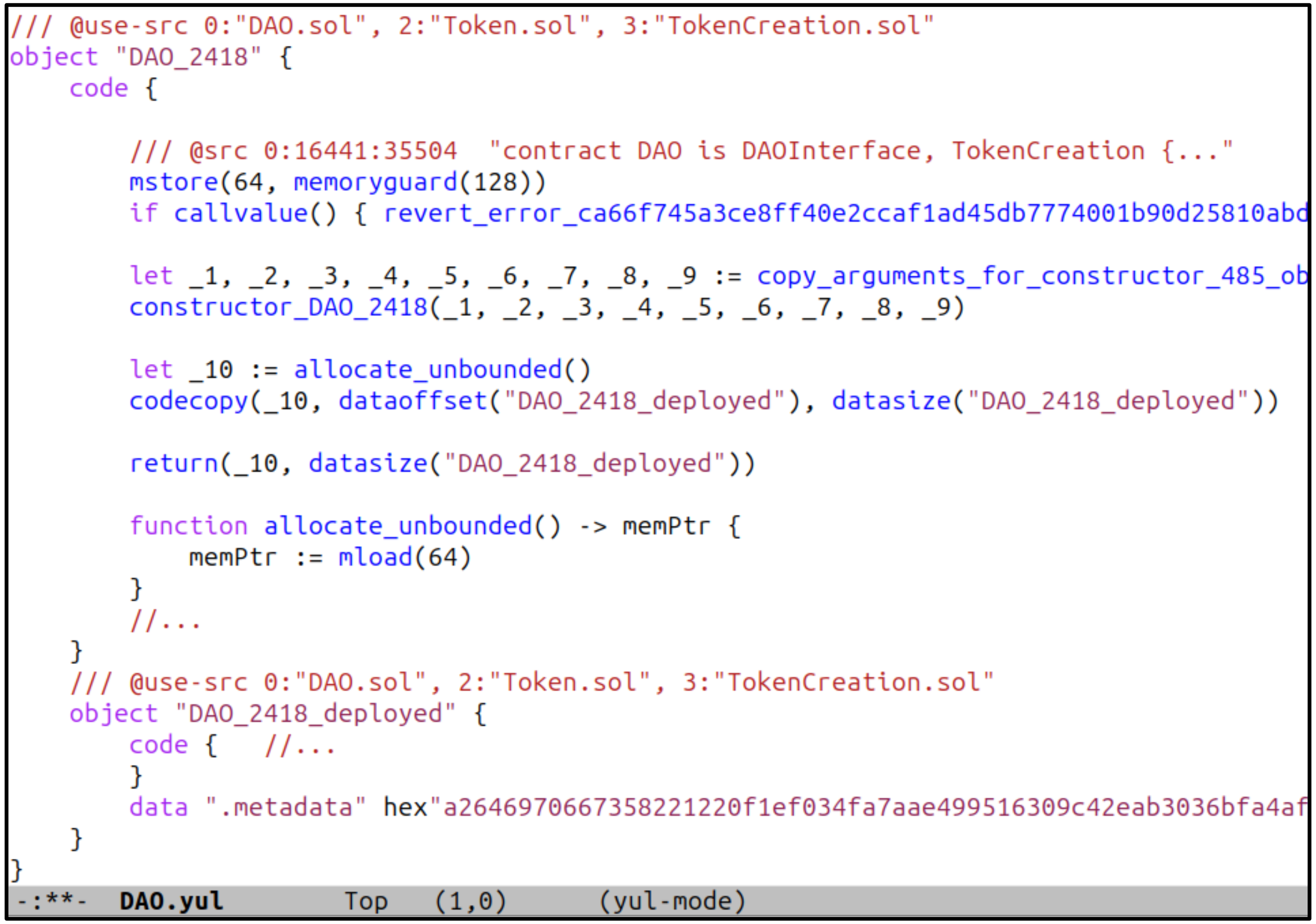
The Yul Language

Introduced by the Solidity Compiler team for readability, optimisation and formal verification

Intermediate representation:

- Replaces stack- and jump- based opcodes with variables and higher-level control flow
- **Target-independent:**
  - Yul only specifies control flow and local state
  - Designed to be modular with regards to so-called dialects
- **Dialects** are defined in relation to a bytecode back-end and consist of:
  - Machine opcodes (e.g. add, mstore, create)
  - Value types (e.g. uint256) used by said opcodes

Example: solc EVM-Flavoured Yul



```
/// @use-src 0:"DAO.sol", 2:"Token.sol", 3:"TokenCreation.sol"
object "DAO_2418" {
    code {

        /// @src 0:16441:35504  "contract DAO is DAOInterface, TokenCreation {..."
        mstore(64, memoryguard(128))
        if callvalue() { revert_error_ca66f745a3ce8ff40e2ccaf1ad45db7774001b90d25810abd

        let _1, _2, _3, _4, _5, _6, _7, _8, _9 := copy_arguments_for_constructor_485_ob
        constructor_DAO_2418(_1, _2, _3, _4, _5, _6, _7, _8, _9)

        let _10 := allocate_unbounded()
        codecopy(_10, dataoffset("DAO_2418_deployed"), datasize("DAO_2418_deployed"))

        return(_10, datasize("DAO_2418_deployed"))

        function allocate_unbounded() -> memPtr {
            memPtr := mload(64)
        }
        //...
    }
    /// @use-src 0:"DAO.sol", 2:"Token.sol", 3:"TokenCreation.sol"
    object "DAO_2418_deployed" {
        code {   //...
        }
        data ".metadata" hex"a2646970667358221220f1ef034fa7aae499516309c42eab3036bfa4af
    }
}
-:**-  DAO.yul        Top    (1,0)        (yul-mode)
```

- Organised into objects
- Object semantics officially unspecified
- Yul specifies only the code fragment
- Without a dialect, Yul consists:
  - Blocks
  - Function calls
  - Conditional branching
  - Loops
  - Variable manipulation

Our Interest: Verifying Smart Contracts

Our previous work: automated verification, bisimilarity of higher-order programs

- TACAS 2022, LICS 2023, LICS 2024

Why Verify Ethereum Smart Contracts?

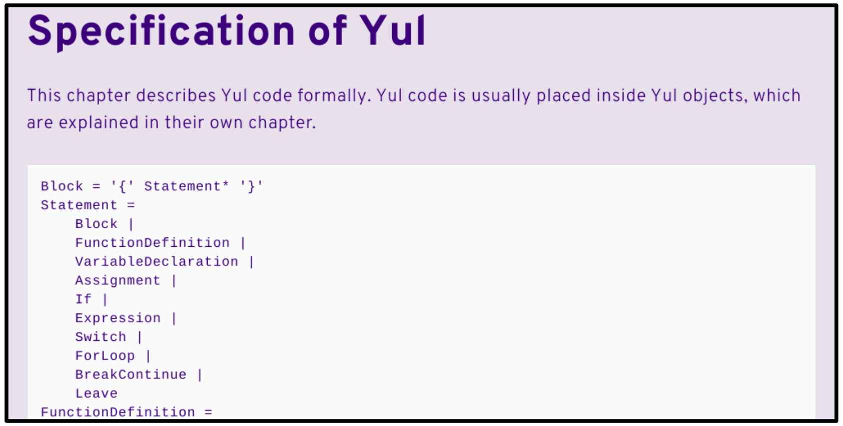
- Immutable, often targetted and high-valued
- Subtle higher-order behaviours (e.g. reentrancy)
- The Ethereum community has recognised that formal verification is important

Yul is a good candidate language:

- Goals of the language include formal verification
- Keeps more information about the original structure of the smart contract than bytecode
- Much simpler and more stable in syntax and features than Solidity

However, **no tool-independent formal specification**

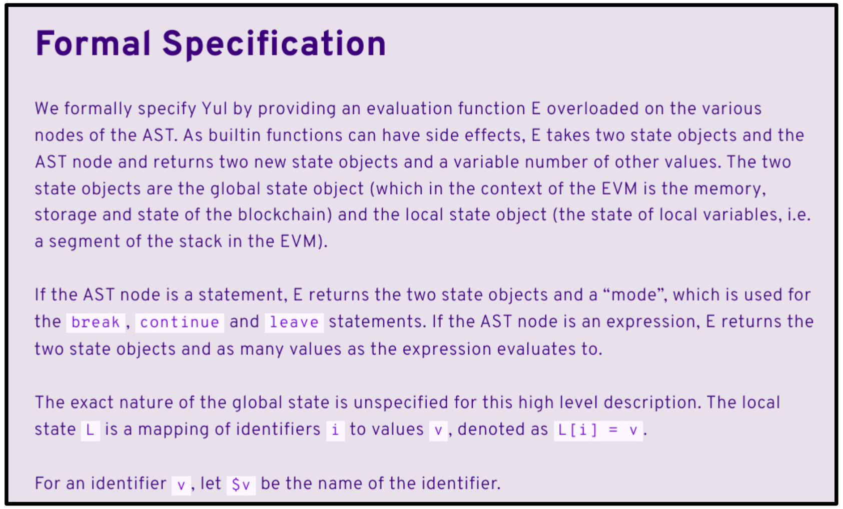
Official Documentation



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 =
```



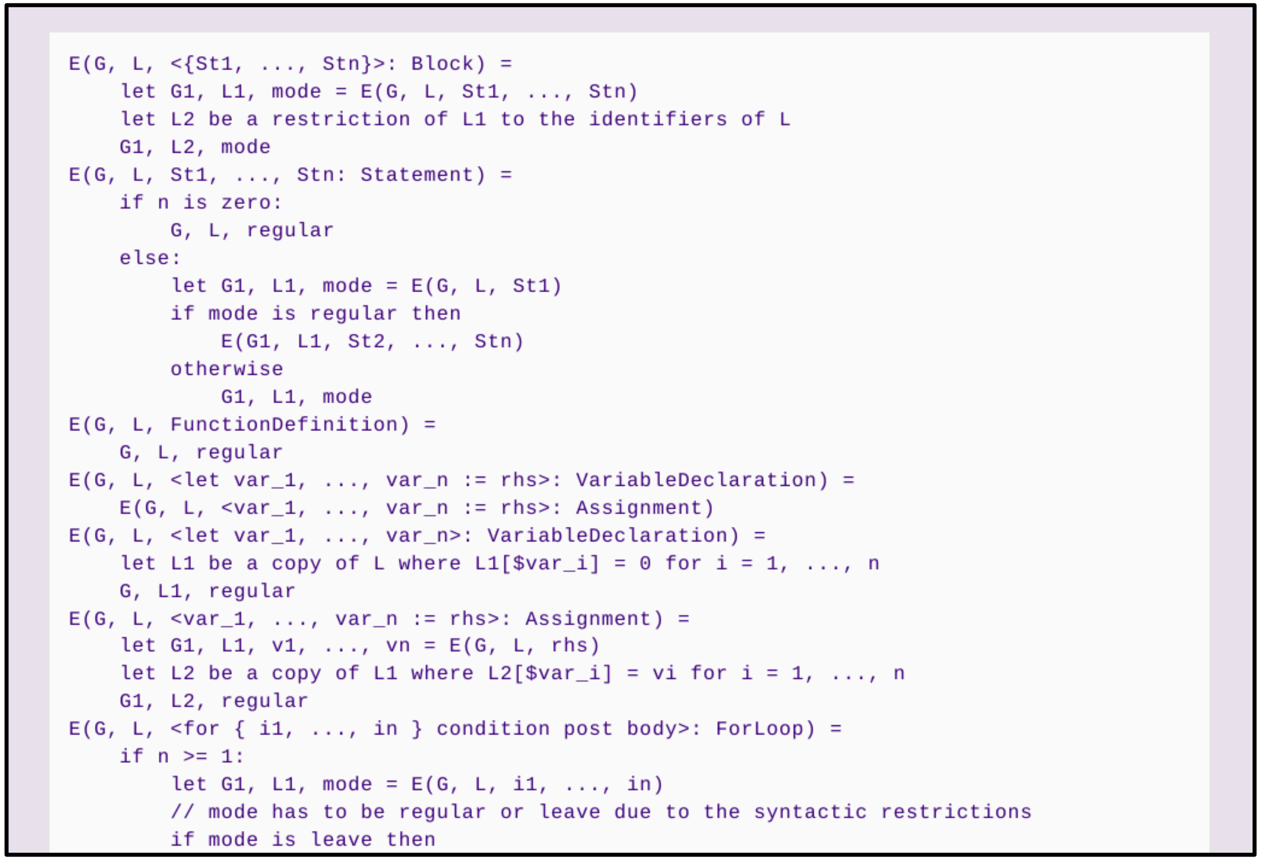
Formal Specification

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.



```
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
```

English and pseudocode; concise, but informal

Some Existing Mechanisations

Multiple tool-based formalisations exist:

- **Dafny:** shallow embedding; assumes Dafny semantics are similar to Yul
- **K-framework:** perhaps an early version; appears incomplete
- **Isabelle/HOL:** ~1600 LoC for the Yul fragment
- **Lean:** ~2000 LoC for Yul
- **ACL2:** part of the the Kestrel books in the large ALC2 repository (2.5GB)
    - 223 files for the Yul fragment; 134,154 LoC (all Yul fragment files combined)

Need for Higher-Level Models

While significant contributions, mechanisations have issues as high-level models:

- Require tool expertise
- Often not peer-reviewed or unmaintained
- Hard to compare against each other
- Hard to use for high-level proofs
- Incorporate implementation details, which may affect the presentation:
  - Too verbose to serve as a high-level model
  - Less standard due to engineering choices being baked into the semantics

Often better to test models than to communicate them

Our Contribution

This Paper:

- Big- and small-step semantics in standard mathematical notation
- Proof of equivalence between big- and small-step
- Implementation of the small-step semantics

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v \mid G_1 ; L_1 ; \mathcal{N}\rangle \qquad x \notin \mathsf{dom}(L)}{\langle \mathsf{let}\ x\,{:}{=}M \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1 ; L_1[x \mapsto v] ; \mathcal{N}\rangle}\ \textsc{VarDecl}$$

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle {<}\vec{v}{>} \mid G_1 ; L_1 ; \mathcal{N}\rangle \qquad \vec{x} \notin \mathsf{dom}(L)}{\langle \mathsf{let}\ \vec{x}\,{:}{=}M \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1 ; L_1[\vec{x} \mapsto \vec{v}] ; \mathcal{N}\rangle}\ \textsc{TupleDecl}$$

$$\textit{TDecl} : \langle \mathsf{let}\ \vec{x}\,{:}{=}{<}\vec{v}{>} \mid L ; \mathcal{N}\rangle \to \langle \mathsf{regular} \mid L[\vec{x} \mapsto \vec{v}] ; \mathcal{N}\rangle \ \text{ if } \vec{x} \notin \mathsf{dom}(L)$$

$$\textit{VDecl} : \langle \mathsf{let}\ x\,{:}{=}v \mid L ; \mathcal{N}\rangle \to \langle \mathsf{regular} \mid L[x \mapsto v] ; \mathcal{N}\rangle \qquad \text{ if } x \notin \mathsf{dom}(L)$$

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{ff} \mid G_0 ; L_0 ; \mathcal{N}\rangle}{\langle \mathsf{if}\ M S \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_0 ; L_0 ; \mathcal{N}\rangle}\ \textsc{IfFalse}$$

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_0 ; L_0 ; \mathcal{N}\rangle \qquad \langle S \mid G_0 ; L_0 ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}{\langle \mathsf{if}\ M S \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}\ \textsc{IfTrue}$$

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v \mid G_0 ; L_0 ; \mathcal{N}\rangle \qquad \langle S_d \mid G_0 ; L_0 ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}{\langle \mathsf{switch}\ M C_1..C_n \mathsf{default} S_d \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}\ \textsc{SwD}$$

$$\frac{\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle c_{n+1} \mid G_0 ; L_0 ; \mathcal{N}\rangle \qquad \langle S_{n+1} \mid G_0 ; L_0 ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}{\langle \mathsf{switch}\ M C_1..C_{n+1}..\mathsf{default} S_d \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1 ; L_1 ; \mathcal{N}\rangle}\ \textsc{SwC}$$

$$\textit{IfFalse} : \langle \mathsf{if\ ff}\ S \mid L ; \mathcal{N}\rangle \to \langle \mathsf{regular} \mid L ; \mathcal{N}\rangle \qquad \textit{IfTrue} : \langle \mathsf{if\ tt}\ S \mid L ; \mathcal{N}\rangle \to \langle S \mid L ; \mathcal{N}\rangle$$

$$\textit{SwDeflt} : \langle \mathsf{switch}\ v\, C_1..C_n \mathsf{default}\ S_d \mid L ; \mathcal{N}\rangle \to \langle S_d \mid L ; \mathcal{N}\rangle$$

$$\textit{SwCase} : \langle \mathsf{switch}\ c_{n+1}\, C_1..C_{n+1}..\mathsf{default}\ S_d \mid L ; \mathcal{N}\rangle \to \langle S_{n+1} \mid L ; \mathcal{N}\rangle$$

Our Formalisation: Source-Level Syntax

Stmt:	$S ::= \{S^*\}$	Blocks
	$\mid \textsf{function}\ x\,(\vec{x}) \to \vec{x}\{S^*\}$	Function Definitions
	$\mid \textsf{let}\ \vec{x} \texttt{:=} M \mid \vec{x} \texttt{:=} M$	Assignment
	$\mid \textsf{if}\ M\{S^*\} \mid \textsf{switch}\ M(\textsf{case}\ v\{S^*\})^*\textsf{default}\{S^*\}$	Branching
	$\mid \textsf{for}\{S^*\}M\{S^*\}\{S^*\}$	Loops
	$\mid \textsf{break} \mid \textsf{continue} \mid \textsf{leave}$	Halting Statements
	$\mid M$	
Exp:	$M ::= x\,(\vec{M})$	Function Call
	$\mid op\ (\vec{M})$	Opcode Call
	$\mid x \mid v$	Variables and Values

Val: u, v, c Var: x, y, z, f

Our Formalisation: Semantics

Configurations:

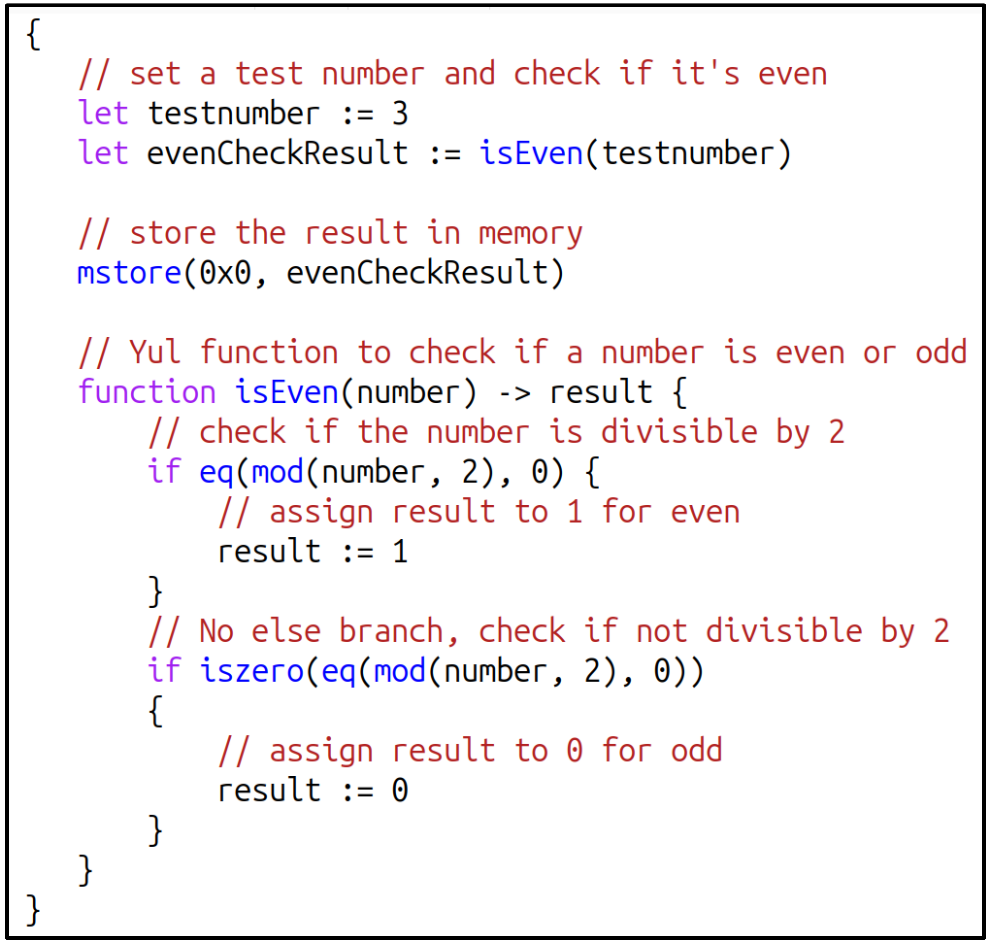
$$\langle S \mid G\,;L\,;\mathcal{N} \rangle$$

- **Statement** S (or *term*) being evaluated
- **Global environment** G
- **Local state** L of all variables. $L : x \mapsto v$.
- **Namespace** $\mathcal{N}$ of all functions visible to the term. $\mathcal{N} : x \mapsto (\vec{x}_a, \vec{x}_r, S_b)$.

Operational Semantics:

- **5 categories of rules:** blocks, variables, branching, loops, expressions
- **Big-Step:** 19 rules combined
- **Small-Step:** 21 rules combined

Example: Parity Check



```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Initial Settings:

- Empty local state and namespace
    - $L = \emptyset$, $\mathcal{N} = \emptyset$
- Global component G is the EVM blockchain
    - Assume some memory component
        - $G.\mathrm{mem} = \emptyset$
- Assume some evaluating term with context E

Note: This example is only illustrative

- Redundant else-branch: `result` is already 0
- Accessing memory in the EVM requires an address
- We shall skip operational syntax (refer to paper)

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \emptyset$
- $\mathcal{N} = \emptyset$
- $E =$
  - $[\ .\ ]\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \varnothing$
- $L = \varnothing$
- $\mathcal{N} = \text{isEven} \rightarrow (\text{number}, \text{result}, \text{body})$
- $E =$
  - $[\ .\ ] \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \mathrm{testnumber} \to 3$
- $\mathcal{N} = \mathrm{isEven} \to (\mathrm{number}, \mathrm{result}, \mathrm{body})$
- $E =$
  - $\mathrm{testnumber} := 3 \ldots$

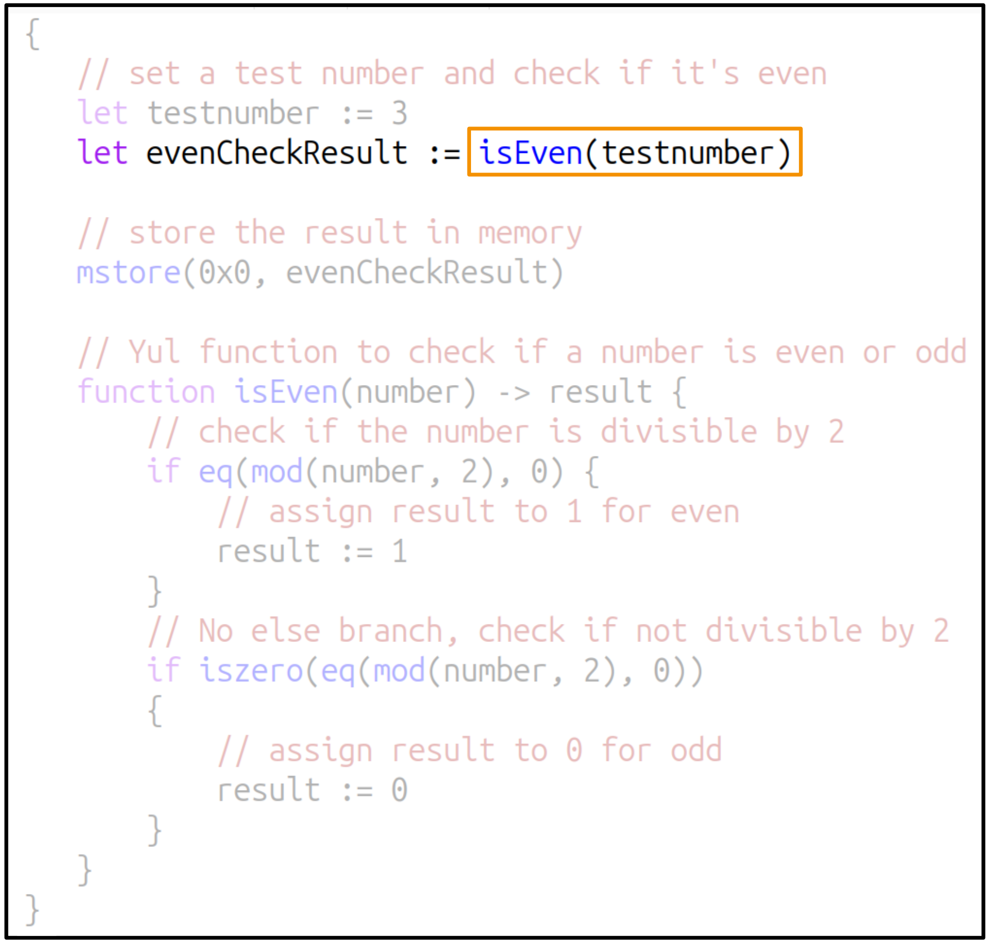
Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \varnothing$
- $L = \mathrm{testnumber} \rightarrow 3$
- $\mathcal{N} = \mathrm{isEven} \rightarrow (\mathrm{number}, \mathrm{result}, \mathrm{body})$
- $E =$
  - $\mathrm{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check



```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \varnothing$
- $L = \text{testnumber} \rightarrow 3$
- $\mathcal{N} = \text{isEven} \rightarrow (\text{number}, \text{result}, \text{body})$
- $E =$
  - $\text{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check

```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \to 3,\ \text{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $[\ .\ ](L = \text{number} \to 3, \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

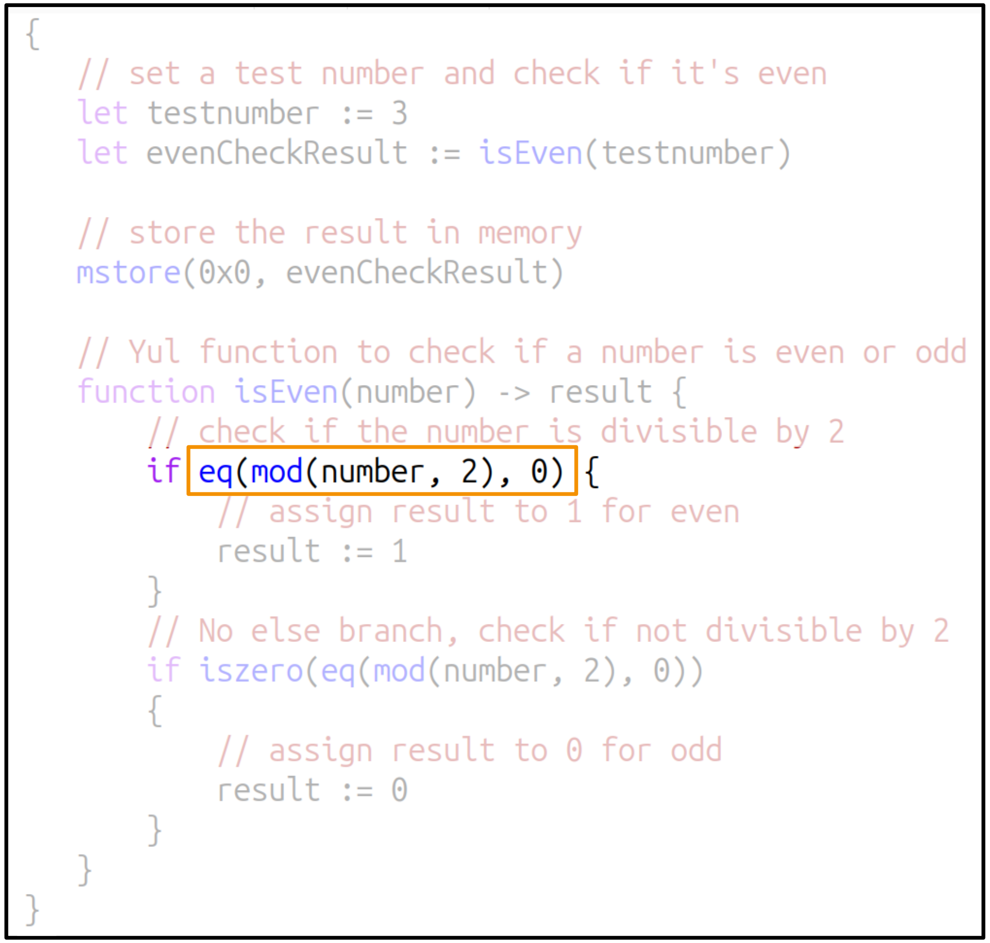
Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
      // check if the number is divisible by 2
      if eq(mod(number, 2), 0) {
         // assign result to 1 for even
         result := 1
      }
      // No else branch, check if not divisible by 2
      if iszero(eq(mod(number, 2), 0))
      {
         // assign result to 0 for odd
         result := 0
      }
   }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \rightarrow 3,\ \text{result} \rightarrow 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{if } [\ .\ ] \ldots$
  - $[\ .\ ](L = \text{number} \rightarrow 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check

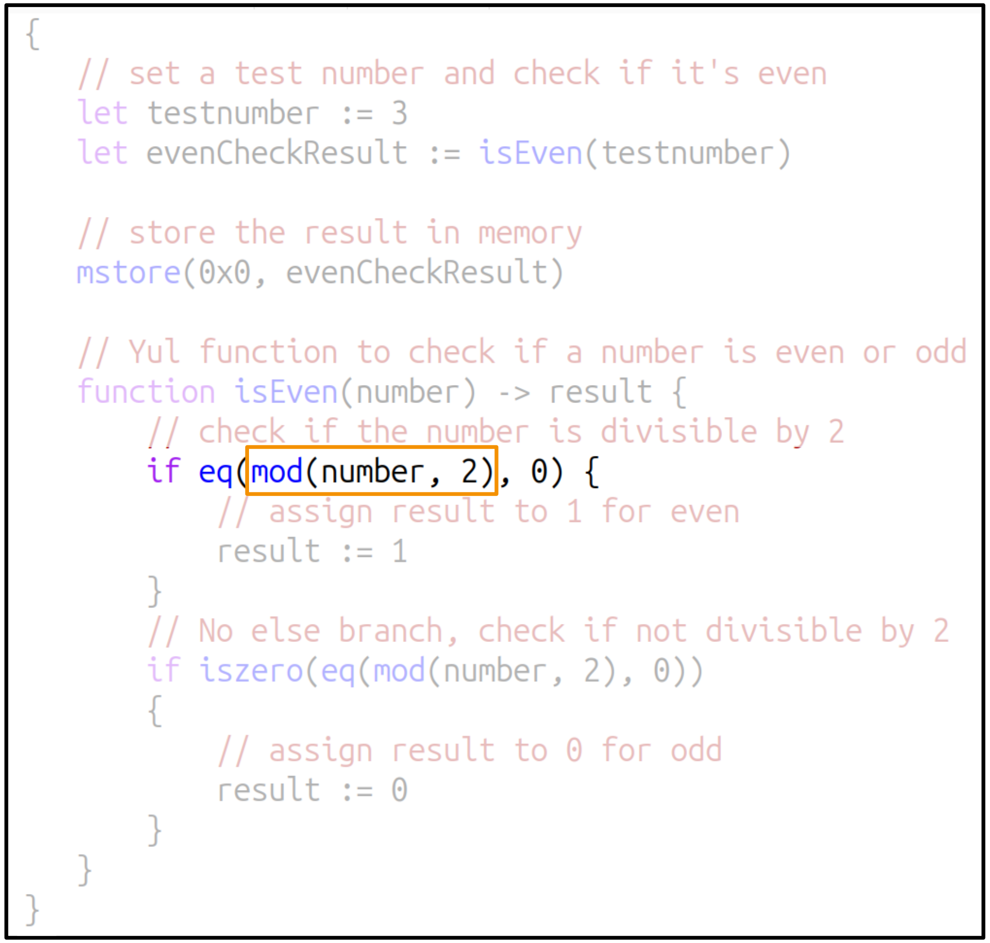


```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \rightarrow 3,\ \text{result} \rightarrow 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{eq}(\ [\ .\ ]\ ,\ 0\ )$
  - $\text{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \text{number} \rightarrow 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check



```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \text{number} \to 3,\ \text{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\mathrm{mod}(3,\ 2)$
  - $\mathrm{eq}(\ [\ .\ ]\ ,\ 0\ )$
  - $\text{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \text{number} \to 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

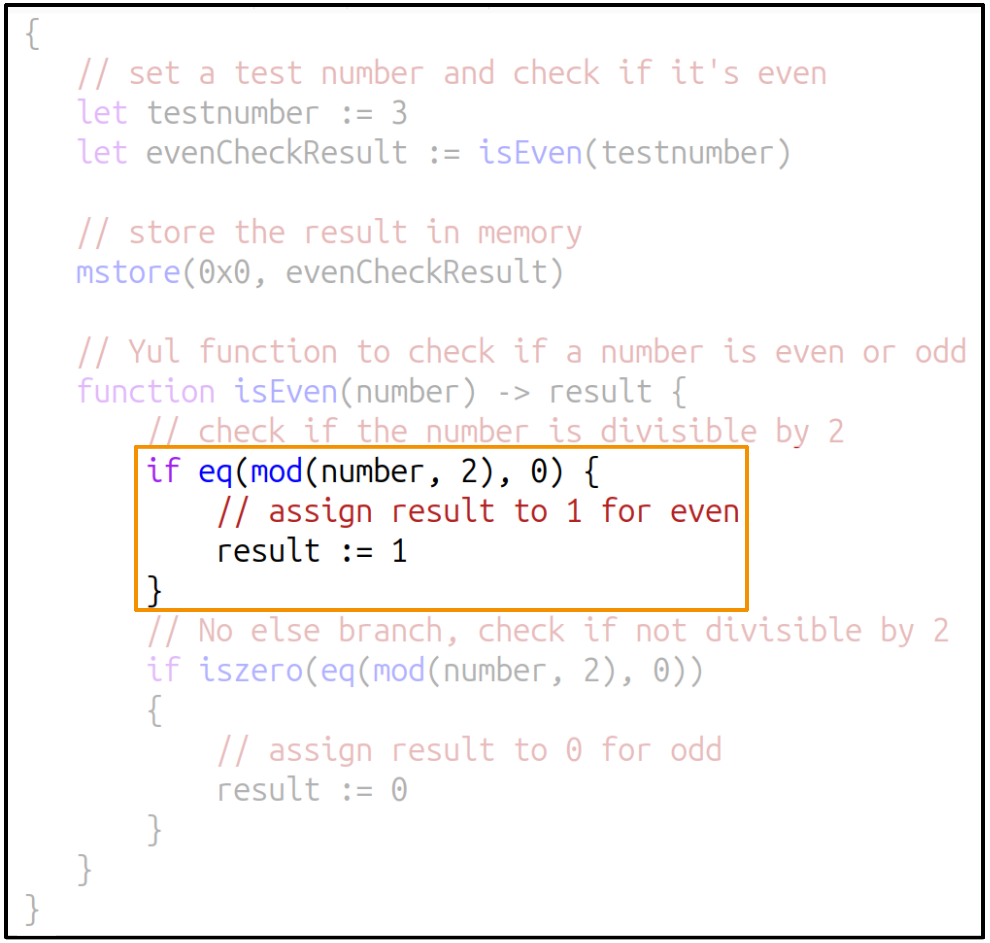
Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \mathrm{number} \to 3,\ \mathrm{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\mathrm{eq}(\ 1\ ,\ 0\ )$
  - $\mathrm{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \mathrm{number} \to 3, \mathcal{N} = \ldots)\ \ldots$
  - $\mathrm{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check



```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \varnothing$
- $L = \text{number} \to 3, \text{ result} \to 0$
- $\mathcal{N} = \varnothing$
- $E =$
    - $\text{if } 0 \ldots$
    - $[\ .\ ](L = \text{number} \to 3, \mathcal{N} = \ldots) \ldots$
    - $\text{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check

```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \rightarrow 3, \text{ result} \rightarrow 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{if } [\ .\ ] \dots$
  - $[\ .\ ](L = \text{number} \rightarrow 3, \mathcal{N} = \dots) \dots$
  - $\text{evenCheckResult} := [\ .\ ] \dots$

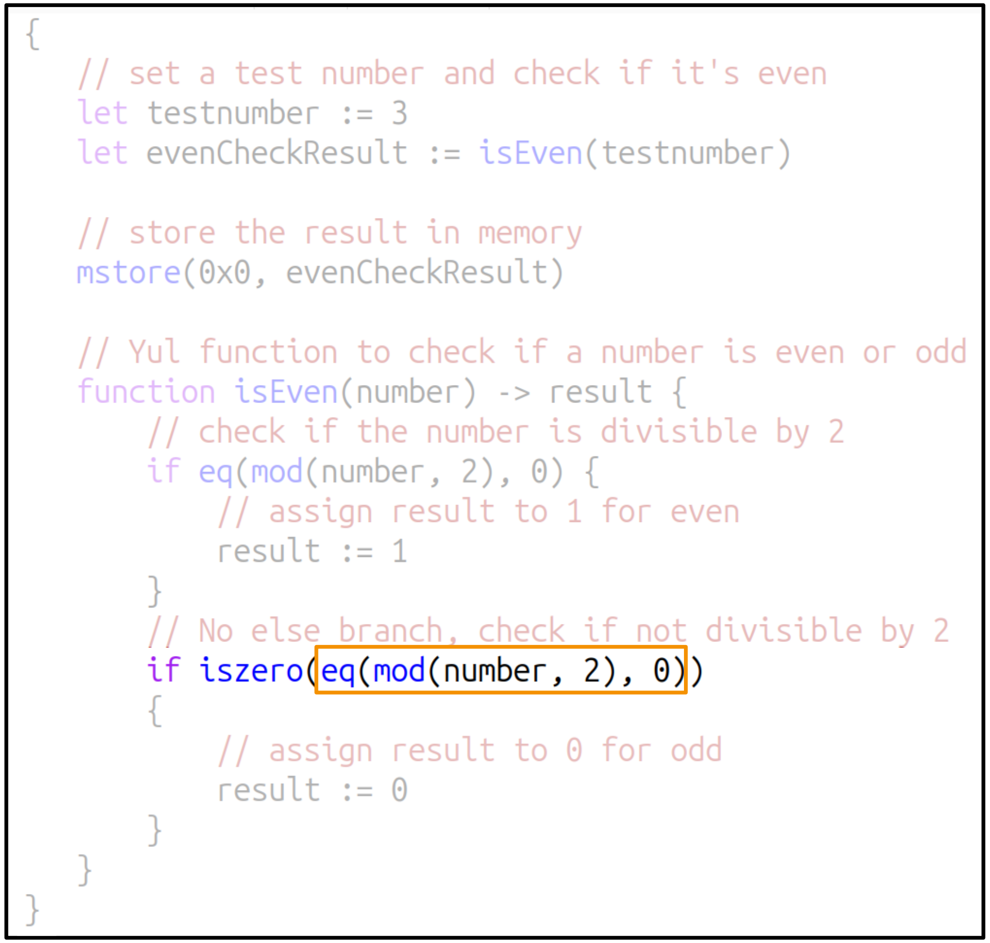
Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \varnothing$
- $L = \text{number} \to 3, \text{ result} \to 0$
- $\mathcal{N} = \varnothing$
- $E =$
  - $\text{iszero}(\ [\ .\ ]\ )$
  - $\text{if } [\ .\ ] \ldots$
  - $[\ .\ ](L = \text{number} \to 3, \mathcal{N} = \ldots) \ldots$
  - $\text{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check

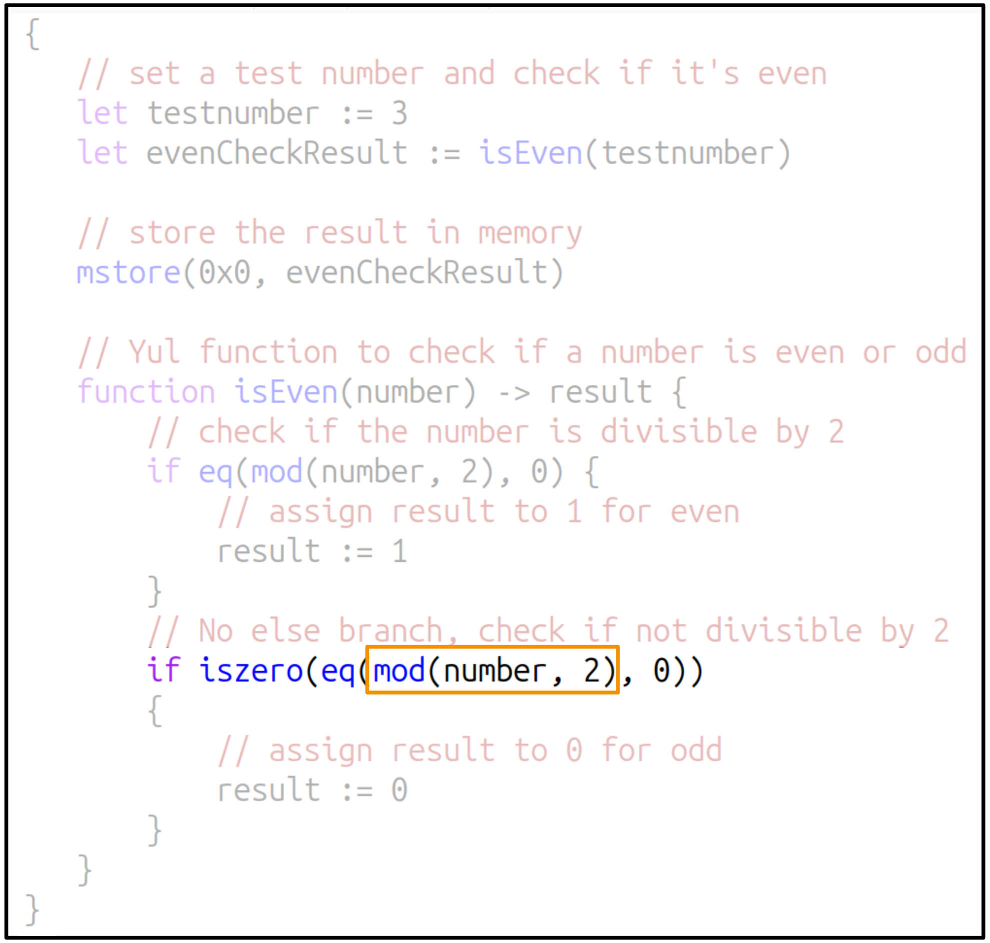


```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\mathrm{mem} = \varnothing$
- $L = \mathrm{number} \rightarrow 3,\ \mathrm{result} \rightarrow 0$
- $\mathcal{N} = \varnothing$
- $E =$
  - $\mathrm{eq}(\ [\ .\ ]\ ,\ 0\ )$
  - $\mathrm{iszero}(\ [\ .\ ]\ )$
  - $\mathrm{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \mathrm{number} \rightarrow 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\mathrm{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check



```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \to 3,\ \text{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{mod}(3,\ 2)$
  - $\text{eq}(\ [\ .\ ]\ ,\ 0\ )$
  - $\text{iszero}(\ [\ .\ ]\ )$
  - $\text{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \text{number} \to 3,\ \mathcal{N} = \ldots)\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \varnothing$
- $L = \text{number} \rightarrow 3, \text{ result } \rightarrow 0$
- $\mathcal{N} = \varnothing$
- $E =$
  - $\text{eq}(\ 1\ ,\ 0\ )$
  - $\text{iszero}(\ [\ .\ ]\ )$
  - $\text{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \text{number} \rightarrow 3, \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \rightarrow 3,\ \text{result} \rightarrow 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{iszero}(\ 0\ )$
  - $\text{if}\ [\ .\ ]\ \ldots$
  - $[\ .\ ](L = \text{number} \rightarrow 3, \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
      // check if the number is divisible by 2
      if eq(mod(number, 2), 0) {
         // assign result to 1 for even
         result := 1
      }
      // No else branch, check if not divisible by 2
      if iszero(eq(mod(number, 2), 0))
      {
         // assign result to 0 for odd
         result := 0
      }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \text{number} \to 3,\ \text{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $\text{if } 1 \ \ldots$
  - $[\ .\ ](L = \text{number} \to 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\mathrm{mem} = \emptyset$
- $L = \mathrm{number} \to 3,\ \mathrm{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $[\ .\ ]$
  - $[\ .\ ](L = \mathrm{number} \to 3,\ \mathcal{N} = \ldots)\ \ldots$
  - $\mathrm{evenCheckResult} := [\ .\ ]\ \ldots$

Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{number} \to 3,\ \text{result} \to 0$
- $\mathcal{N} = \emptyset$
- $E =$
  - $[\ .\ ](L = \text{number} \to 3, \mathcal{N} = \ldots)\ \ldots$
  - $\text{evenCheckResult} := [\ .\ ]\ \ldots$

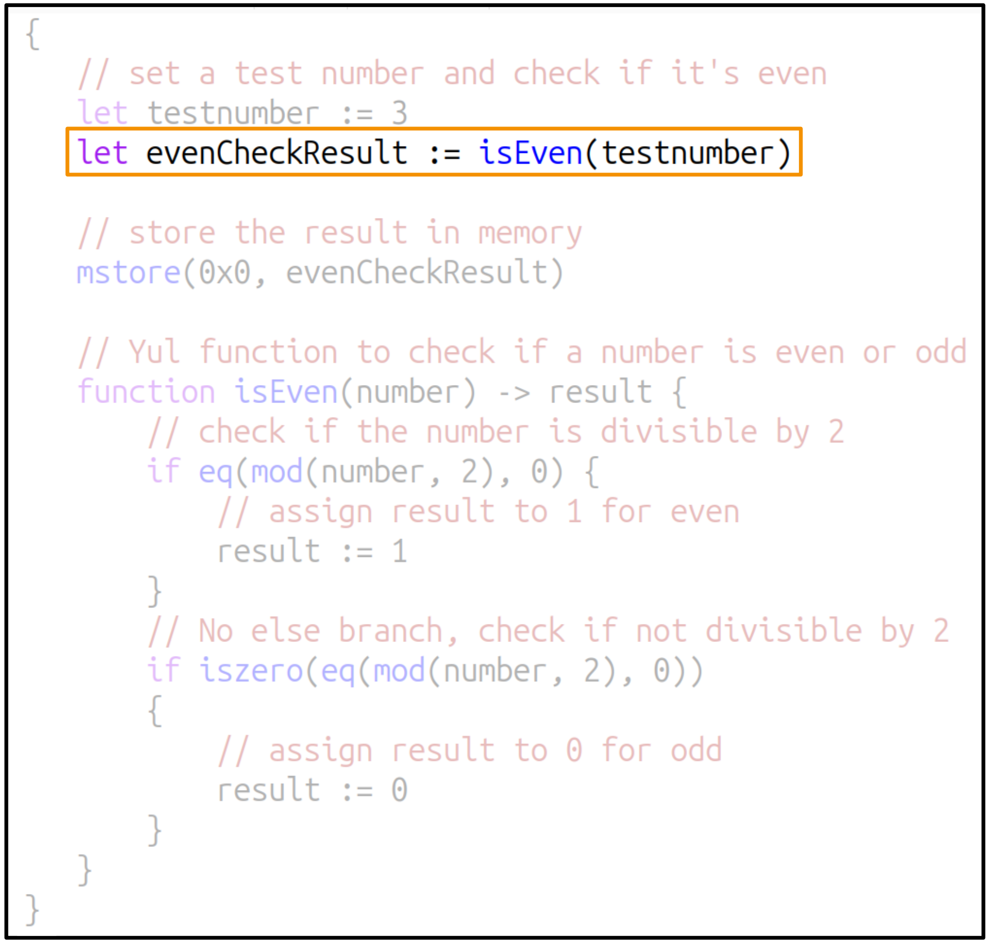
Example: Parity Check

```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{testnumber} \rightarrow 3$
- $\mathcal{N} = \text{isEven} \rightarrow (\text{number}, \text{result}, \text{body})$
- $E =$
  - 0
  - $\text{evenCheckResult} := [\ .\ ] \ldots$

Example: Parity Check



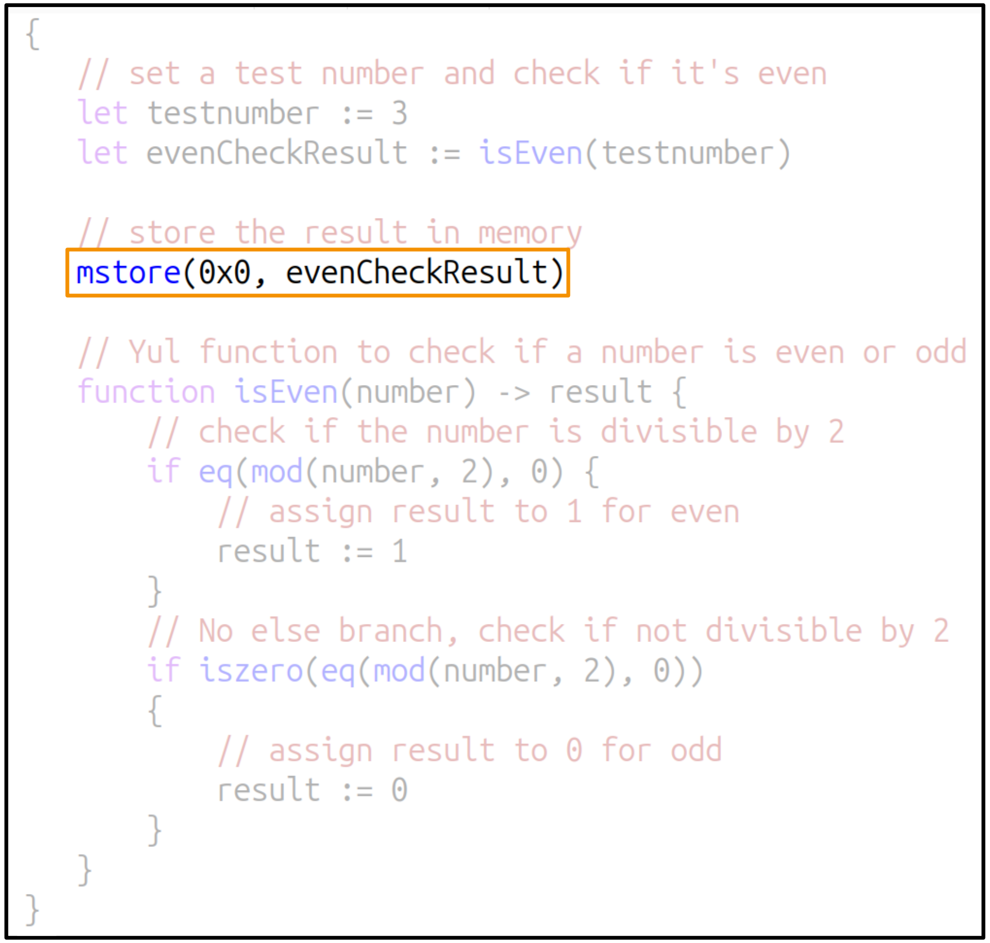
```
{
    // set a test number and check if it's even
    let testnumber := 3
    let evenCheckResult := isEven(testnumber)

    // store the result in memory
    mstore(0x0, evenCheckResult)

    // Yul function to check if a number is even or odd
    function isEven(number) -> result {
        // check if the number is divisible by 2
        if eq(mod(number, 2), 0) {
            // assign result to 1 for even
            result := 1
        }
        // No else branch, check if not divisible by 2
        if iszero(eq(mod(number, 2), 0))
        {
            // assign result to 0 for odd
            result := 0
        }
    }
}
```

Configuration:

- $G.\text{mem} = \emptyset$
- $L = \text{testnumber} \rightarrow 3$

  $\text{evmCheckResult} \rightarrow 0$
- $\mathcal{N} = \text{isEven} \rightarrow (\text{number}, \text{result}, \text{body})$
- $E =$
  - $\text{evenCheckResult} := 0 \ldots$

Example: Parity Check



```
{
   // set a test number and check if it's even
   let testnumber := 3
   let evenCheckResult := isEven(testnumber)

   // store the result in memory
   mstore(0x0, evenCheckResult)

   // Yul function to check if a number is even or odd
   function isEven(number) -> result {
       // check if the number is divisible by 2
       if eq(mod(number, 2), 0) {
           // assign result to 1 for even
           result := 1
       }
       // No else branch, check if not divisible by 2
       if iszero(eq(mod(number, 2), 0))
       {
           // assign result to 0 for odd
           result := 0
       }
   }
}
```

Configuration:

- $G.\text{mem} = 0\text{x}0 \rightarrow 0$
- $L = \text{testnumber} \rightarrow 3$

  $\text{evmCheckResult} \rightarrow 0$
- $\mathcal{N} = \text{isEven} \rightarrow (\text{number, result, body})$
- $E =$
  - $\text{mstore}(0\text{x}0\ ,\ 0)$

Example: Parity Check

Blocks:

$$\textit{BStart} : \langle \{S_1..S_n\} \mid L\,;\mathcal{N}\rangle \rightarrow \langle \{S_1, .., S_n\}_L^{\mathcal{N}} \mid L\,;\mathcal{N} \uplus \mathcal{N}_1\rangle \text{ if } \mathcal{N}_1 = \mathsf{funs}(\{S_1..S_n\}), n > 0$$

$$\textit{BEmpty} : \langle \{\} \mid L\,;\mathcal{N}\rangle \rightarrow \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$$

$$\textit{SReg} : \langle \{\mathsf{regular}\}_L^{\mathcal{N}} \mid L_1\,;\mathcal{N}_1\rangle \rightarrow \langle \mathsf{regular} \mid L_1{\restriction}L\,;\mathcal{N}\rangle$$

Variables:

$$\textit{VDecl} : \langle \mathsf{let}\ x := v \mid L\,;\mathcal{N}\rangle \rightarrow \langle \mathsf{regular} \mid L[x \mapsto v]\,;\mathcal{N}\rangle \qquad \text{if } x \notin \mathsf{dom}(L)$$

Conditionals:

$$\textit{IfTrue} : \langle \mathsf{if\ tt}\ S \mid L\,;\mathcal{N}\rangle \rightarrow \langle S \mid L\,;\mathcal{N}\rangle$$

$$\textit{IfFalse} : \langle \mathsf{if\ ff}\ S \mid L\,;\mathcal{N}\rangle \rightarrow \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$$

Expressions:

$$\textit{Ident} : \langle x \mid L\,;\mathcal{N}\rangle \rightarrow \langle L(x) \mid L\,;\mathcal{N}\rangle \qquad \text{if } x \in L$$

$$\textit{FCall} : \langle f\,(\vec{v}) \mid L\,;\mathcal{N}\rangle \rightarrow \langle (\!| S_b |\!)_L^{\vec{z}} \mid L_f\,;\mathcal{N}\rangle$$

$$\textit{FRet} : \langle (\!|\mathbb{M}|\!)_L^{z_1,\dots,z_m} \mid L'\,;\mathcal{N}\rangle \rightarrow \langle <L'(z_1), .., L'(z_m)> \mid L\,;\mathcal{N}\rangle \text{ if } \mathbb{M} \in \{\mathsf{leave}, \mathsf{regular}\}$$

See paper for full set of rules

Example: Parity Check

$$\frac{\mathcal{N}_1 = \mathsf{funs}(\{S_1..S_n\}) \qquad \langle S_1, .., S_n \mid G\,;L\,;\mathcal{N} \uplus \mathcal{N}_1\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N} \uplus \mathcal{N}_1\rangle}{\langle \{S_1..S_n\} \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1{\restriction}L\,;\mathcal{N}\rangle}\;\textsc{Block}$$

$$\frac{}{\langle \varepsilon \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathsf{regular} \mid G\,;L\,;\mathcal{N}\rangle}\;\textsc{SeqEmpty}$$

$$\frac{\langle S_1 \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad \langle \vec{S} \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle}{\langle S_1, \vec{S} \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle}\;\textsc{SeqR}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad x \notin \mathsf{dom}(L)}{\langle \mathsf{let}\ x \mathrel{\mathsf{:=}} M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1[x \mapsto v]\,;\mathcal{N}\rangle}\;\textsc{VarDecl}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{ff} \mid G_0\,;L_0\,;\mathcal{N}\rangle}{\langle \mathsf{if}\ M S \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_0\,;L_0\,;\mathcal{N}\rangle}\;\textsc{IfFalse}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_0\,;L_0\,;\mathcal{N}\rangle \qquad \langle S \mid G_0\,;L_0\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}{\langle \mathsf{if}\ M S \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\;\textsc{IfTrue}$$

$$\frac{}{\langle x \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle L(x) \mid G\,;L\,;\mathcal{N}\rangle}\;\textsc{Id}$$

$$\frac{\begin{array}{c}\langle M_n \mid G; L; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_n \mid G_1; L_1; \mathcal{N}\rangle \ldots \langle M_1 \mid G_{n-1}; L_{n-1}; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_1 \mid G_n; L_n; \mathcal{N}\rangle \\ \langle S_b \mid G_n\,;L_f\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G''\,;L'\,;\mathcal{N}\rangle \qquad \mathbb{M} \in \{\mathsf{regular}, \mathsf{leave}\}\end{array}}{\langle f\,(M_1, .., M_n) \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle < L'(z_1), .., L'(z_m) > \mid G''\,;L_n\,;\mathcal{N}\rangle}\;\textsc{Fun}$$

See paper for full set of rules

Proof: Semantics Equivalence

Theorem 12 (Semantics Equivalence). *Given statement S in source syntax, statement sequence $\vec{S}$, expression M, global environment G, local state L, and function namespace $\mathcal{N}$, then:*

1. $\langle S \,|\, G\,;L\,;\mathcal{N} \rangle \Downarrow \langle \mathbb{M} \,|\, G'\,;L'\,;\mathcal{N}' \rangle$ *iff* $\langle S \,|\, G\,;L\,;\mathcal{N} \rangle \rightarrow^* \langle \mathbb{M} \,|\, G'\,;L'\,;\mathcal{N}' \rangle$
2. $\langle \vec{S} \,|\, G\,;L\,;\mathcal{N} \rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G'\,;L'\,;\mathcal{N}' \rangle$ *iff* $\langle \vec{S} \,|\, G\,;L\,;\mathcal{N} \rangle \rightarrow^* \langle \mathbb{M} \mid G'\,;L'\,;\mathcal{N}' \rangle$
3. $\langle M \mid G\,;L\,;\mathcal{N} \rangle \Downarrow_{\mathsf{exp}} \langle S' \mid G'\,;L'\,;\mathcal{N}' \rangle$ *iff* $\langle M \mid G\,;L\,;\mathcal{N} \rangle \rightarrow^* \langle S' \mid G'\,;L'\,;\mathcal{N}' \rangle$

where $S' \in \{v, <\vec{v}>, \mathsf{regular}\}$.

Implementation

https://github.com/LaifsV1/YulTracer

YulTracer

DOI 10.5281/zenodo.12511493 License MIT

- Currently an interpreter for Yul
  - **WIP:** Symbolic execution tool
- Modular on dialects
- Implements our Yul semantics and a subset of EVM Shanghai
- Written in OCaml, calls Z3 to solve constraints
- Tested against implementations of standard sequence generators and sorting algorithms
- Tested against the Yul tests in the Solidity Compiler repository

Future Work

Goal: Verify Smart Contracts compiled into Yul

Extend YulTracer with full Symbolic Execution:

- Symbolic model for first-order EVM opcodes
    - e.g. Arithmetic, Memory and Storage, Keccak, etc.
- Symbolic model for higher-order EVM opcodes
    - e.g. create, call, etc.
    - Symbolic Game Semantics for EVM-Yul objects
    - Novel finitisation techniques for feasibility
        - e.g. bisimulation up-to techniques