B8: Arrays and Loops

CS1101S: Programming Methodology

Low Kok Lim

October 8, 2021

Outline

- Arrays
- Loops
- Arrays and Loops
- Environments of Arrays and Loops

Outline

- Arrays
- Loops
- Arrays and Loops
- Environments of Arrays and Loops

Arrays

- An ***array*** is a data structure that stores a sequence of data elements

```
const seq = [10, 5, 8];  // array of length 3
let my_array = [];       // empty array
```

- ***Array access*** — each data element can be accessed by using the array's name and a *non-negative integer* ***index***
  - The **first element** has index **0**

```
seq[0]; ➔ 10
seq[2]; ➔ 8
```

Arrays

- ***Array assignment*** — each data element can be assigned to with new value

```
seq[0] = 20;
seq[0]; ➔ 20
```

- Arrays support ***random access***
  - Any element in an array can be **accessed or assigned to** in **constant time**

Array Length

- The primitive function **array_length** returns the length of an array

```
array_length(seq); ➔ 3
array_length(my_array); ➔ 0
```

- The length of an array can be increased by assigning to index position beyond the “last element”

```
seq[10] = 99;
seq[10]; ➔ 99
array_length(seq); ➔ 11
```

Array Example

```
const things = [123, "cat", "orange"];
things;     ➔ [123, "cat", "orange"]
array_length(things); ➔ 3
things[0]; ➔ 123
things[2]; ➔ "orange"
things[2] = "apple";
things[2]; ➔ "apple"
things[4] = 456;
array_length(things); ➔ 5
things;     ➔ [123, "cat", "apple", undefined, 456]
things[4]; ➔ 456
things[3]; ➔ undefined
```

Another Array Example

```
let my_array = [];  // creates an empty array

array_length(my_array); ➔ 0

my_array[5] = 100;

my_array; ➔ [undefined, undefined, undefined,
              undefined, undefined, 100]

array_length(my_array); ➔ 6
```

“Two-Dimensional” Array Example

```
let table = [[1,  2,  3,  4],
             [5,  6,  7,  8],
             [9, 10, 11    ]];

array_length(table); ➔ 3

table[1][2]; ➔ 7

array_length(table[0]); ➔ 4

array_length(table[2]); ➔ 3
```

Processing Arrays — array_1_to_n

```
// array_1_to_n(n) returns an array that
//   contains elements 1 thru n.
function array_1_to_n(n) {
    const a = [];
    function iter(i) {
        if (i < n) {
            a[i] = i + 1;
            iter(i + 1);
        }
    }
    iter(0);
    return a;
}
array_1_to_n(3);  // [1, 2, 3]
```

Show in Playground

Processing Arrays — map_array

```
function map_array(f, arr) {
    const len = array_length(arr);
    function iter(i) {
        if (i < len) {
            arr[i] = f(arr[i]);
            iter(i + 1);
        }
    }
    iter(0);
}

const seq = [3, 1, 5];
map_array(x => 2 * x, seq);
seq; // [6, 2, 10];  destructive operation
```

Show in Playground

Outline

- Arrays
- Loops
- Arrays and Loops
- Environments of Arrays and Loops

`while` Loop

- **Syntax:**

```
while (expression) {
    statement
}
```

- Evaluates **condition expression** *`expression`* and if the result is `true`, executes the body *`statement`* of the loop, after which the process **repeats**. The loop **terminates** when the condition expression evaluates to `false`.

Factorial Using while Loop

```
function factorial_r(n) {
    return (n === 1) ? 1 : n * factorial_r(n - 1);
}
```

```
function factorial_i(n) {
    function f(acc, k) {
        if (k <= n) {
            return f(acc * k,
                     k + 1);
        } else {
            return acc;
        }
    }
    return f(1, 1);
}
```

```
function factorial_w(n) {
    let acc = 1;
    let k = 1;
    while (k <= n) {
        acc = acc * k;
        k = k + 1;
    }
    return acc;
}
```

Show in Playground

for Loop

- **Syntax:**

```
for (stmt1; expression; assignment) {
    statement
}
```

- **Equivalent to**

```
{
    stmt1;
    while (expression) {
        statement
        assignment;
    }
}
```

Note:
This is only a simplified translation/view of the **for**-loop.

For accurate description, please refer to the Source §3 specifications.

Environment model for **for**-loop will not be in assessments.

for Loop

- **Syntax:**

```
for (stmt1; expression; assignment) {
    statement
}
```

- *stmt1;* can only be
  - an **assignment statement** or
  - a **variable declaration statement** (e.g. `let x = 1;`)
    - The variable is called a ***loop control variable***

Restrictions on Loops in Source §3

- The declared **loop control variable** for a `for` loop cannot be assigned to in the body

- All components in the header of a `for` loop are non-optional
  - For example, `for (;;) {...}` is not allowed

Factorial Using for Loop

```
function factorial_f(n) {
    let acc = 1;
    for (let k = 1; k <= n; k = k + 1) {
        acc = acc * k;
    }
    return acc;
}
```

```
function factorial_w(n) {
    let acc = 1;
    let k = 1;
    while (k <= n) {
        acc = acc * k;
        k = k + 1;
    }
    return acc;
}
```

Show in Playground

List Length

```
function list_length(xs) {
    return is_null(xs) ? 0 : 1 + list_length(tail(xs));
}
```

```
function list_length_loop(xs) {
    let count = 0;
    for (let p = xs; !is_null(p); p = tail(p)) {
        count = count + 1;
    }
    return count;
}
```

Show in Playground

The **break;** Statement

- **break;** terminates the current execution of the loop and also terminates the entire loop

```
for (let i = 1; i < 5; i = i + 1) {
    display(stringify(i) + " here");
    if (i === 2) {
        break;
    }
    display(stringify(i) + " there");
}
display("OK");
```

```
Output:
"1 here"
"1 there"
"2 here"
"OK"
```

Show in Playground

The `continue;` Statement

- `continue;` terminates the **current** execution of the loop and continues with the loop

```
for (let i = 1; i < 5; i = i + 1) {
    display(stringify(i) + " here");
    if (i === 2) {
        continue;
    }
    display(stringify(i) + " there");
}
display("OK");
```

```
Output:
"1 here"
"1 there"
"2 here"
"3 here"
"3 there"
"4 here"
"4 there"
"OK"
```

Show in Playground

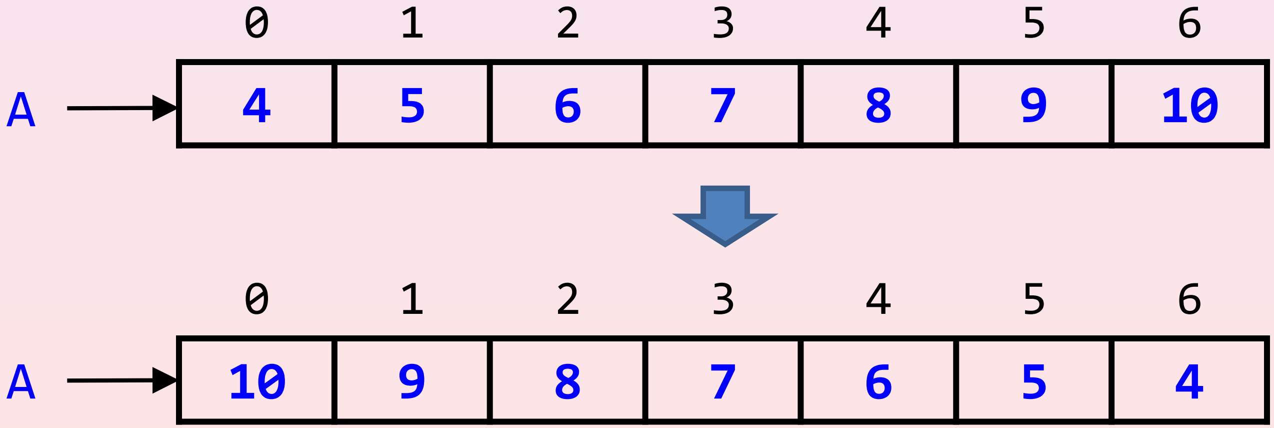
Outline

- Arrays
- Loops
- Arrays and Loops
- Environments of Arrays and Loops

Loops and Arrays — `reverse_array`

- **Wanted:** A `reverse_array` function to reverse the input array
- **Example:**

```
const A = [4, 5, 6, 7, 8, 9, 10];
reverse_array(A);
A; ➔ [10, 9, 8, 7, 6, 5, 4]
```

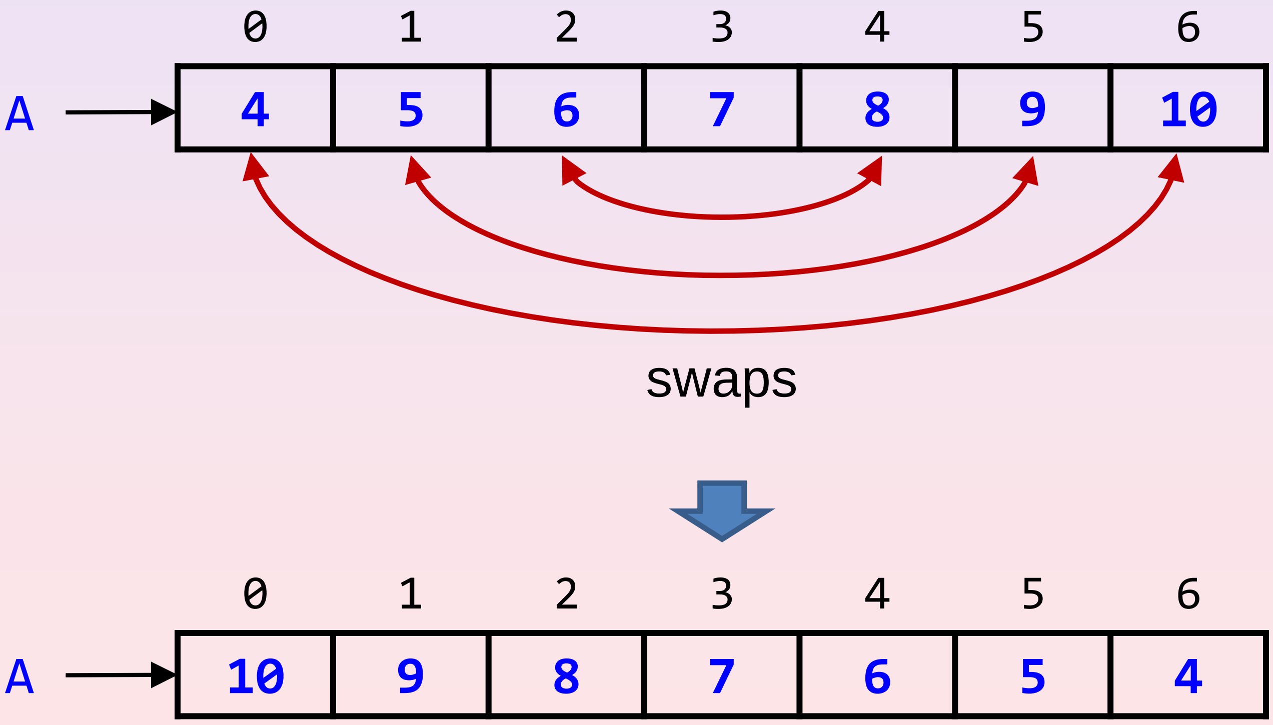


	0	1	2	3	4	5	6
A →	4	5	6	7	8	9	10

	0	1	2	3	4	5	6
A →	10	9	8	7	6	5	4

Loops and Arrays — `reverse_array`

- **How to reverse?**



	0	1	2	3	4	5	6
A →	4	5	6	7	8	9	10

swaps

	0	1	2	3	4	5	6
A →	10	9	8	7	6	5	4

Loops and Arrays — `reverse_array (Attempt #1)`

- **Attempt #1:**

```
function swap(x, y) {
    let temp = x;
    x = y;
    y = temp;
}

function reverse_array(A) {
    const len = array_length(A);
    const half_len = math_floor(len / 2);
    for (let i = 0; i < half_len; i = i + 1) {
        swap(A[i], A[len - 1 - i]);
    }
}
```

Show in Playground

Loops and Arrays — `reverse_array (Attempt #1)`

- **Testing:**

```
const A = [4, 5, 6, 7, 8, 9, 10];
reverse_array(A);
A; ➔ [4, 5, 6, 7, 8, 9, 10]
```

- **What is wrong?**

Loops and Arrays — reverse_array (Attempt #2)

- **Attempt #2:**

```
function swap(A, i, j) {
    let temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

function reverse_array(A) {
    const len = array_length(A);
    const half_len = math_floor(len / 2);
    for (let i = 0; i < half_len; i = i + 1) {
        swap(A, i, len - 1 - i);
    }
}
```

Show in Playground

Loops and Arrays — zero_matrix

```
// Returns a 2D array that represents
//   a rows x cols zero matrix.
function zero_matrix(rows, cols) {
    const M = [];
    for (let r = 0; r < rows; r = r + 1) {
        M[r] = [];
        for (let c = 0; c < cols; c = c + 1) {
            M[r][c] = 0;
        }
    }
    return M;
}

const mat3x4 = zero_matrix(3, 4);
```

Show in Playground

Loops and Arrays — matrix_multiply_3x3

```
// Returns a 2D array represents the results
//   of multiplying two 3x3 matrices.
function matrix_multiply_3x3(A, B) {
    const M = [];
    for (let r = 0; r < 3; r = r + 1) {
        M[r] = [];
        for (let c = 0; c < 3; c = c + 1) {
            M[r][c] = 0;
            for (let k = 0; k < 3; k = k + 1) {
                M[r][c] = M[r][c] + A[r][k] * B[k][c];
            }
        }
    }
    return M;
}
```

Show in Playground

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} * \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix}$$

Outline

- Arrays
- Loops
- Arrays and Loops
- Environments of Arrays and Loops

`while` Loop

- **Syntax:**

```
while (expression) {
    statement
}
```

- The **loop body** is in a **new block** (`{ statement }`)

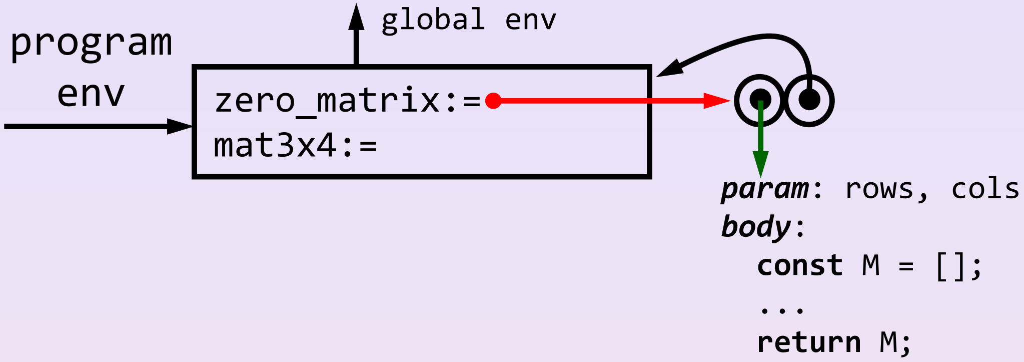
- ***Every time*** when the **body block** is evaluated, it extends the environment by adding a **new frame**
    - **No new frame** is created if the block has **no constant & variable declaration**

Environments of Loops and Arrays: Example

```
// Using while loops
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}

const mat3x4 = zero_matrix(3, 4);
```

Show in Playground



global env

program env

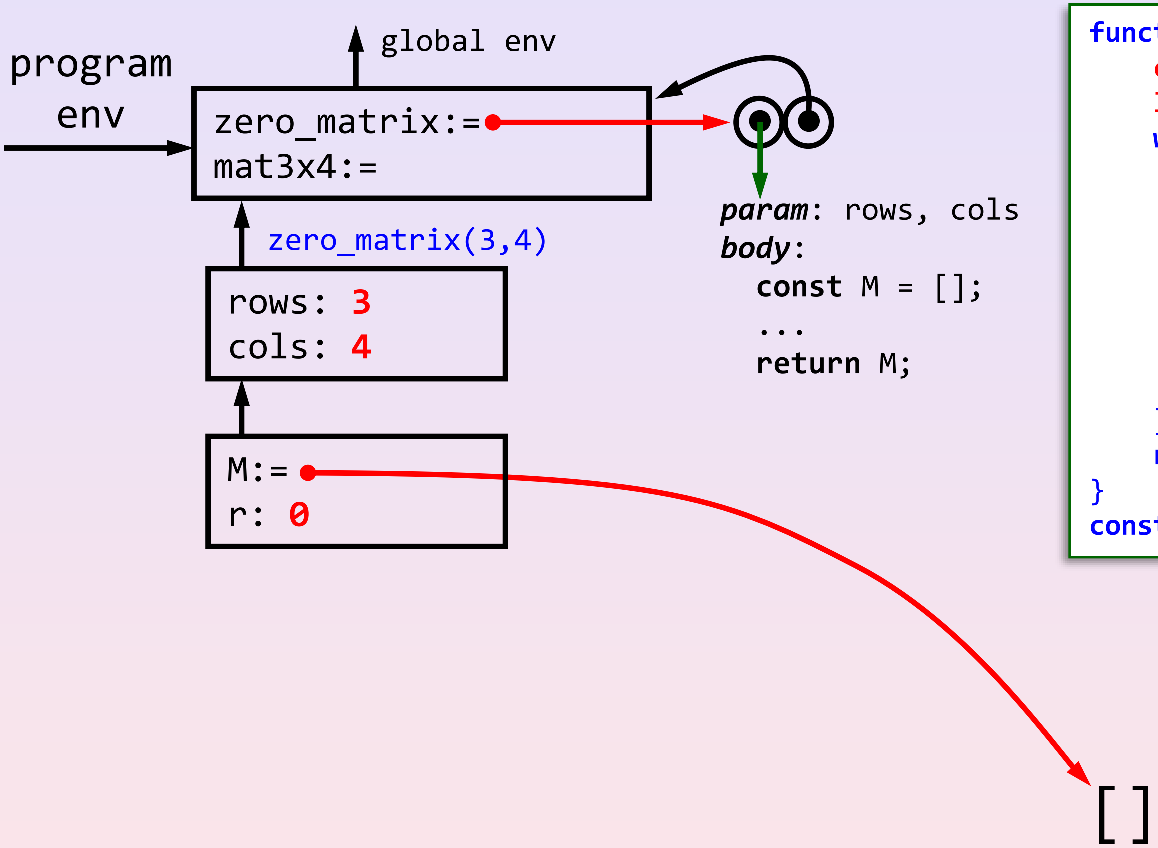
zero_matrix:=

mat3x4:=

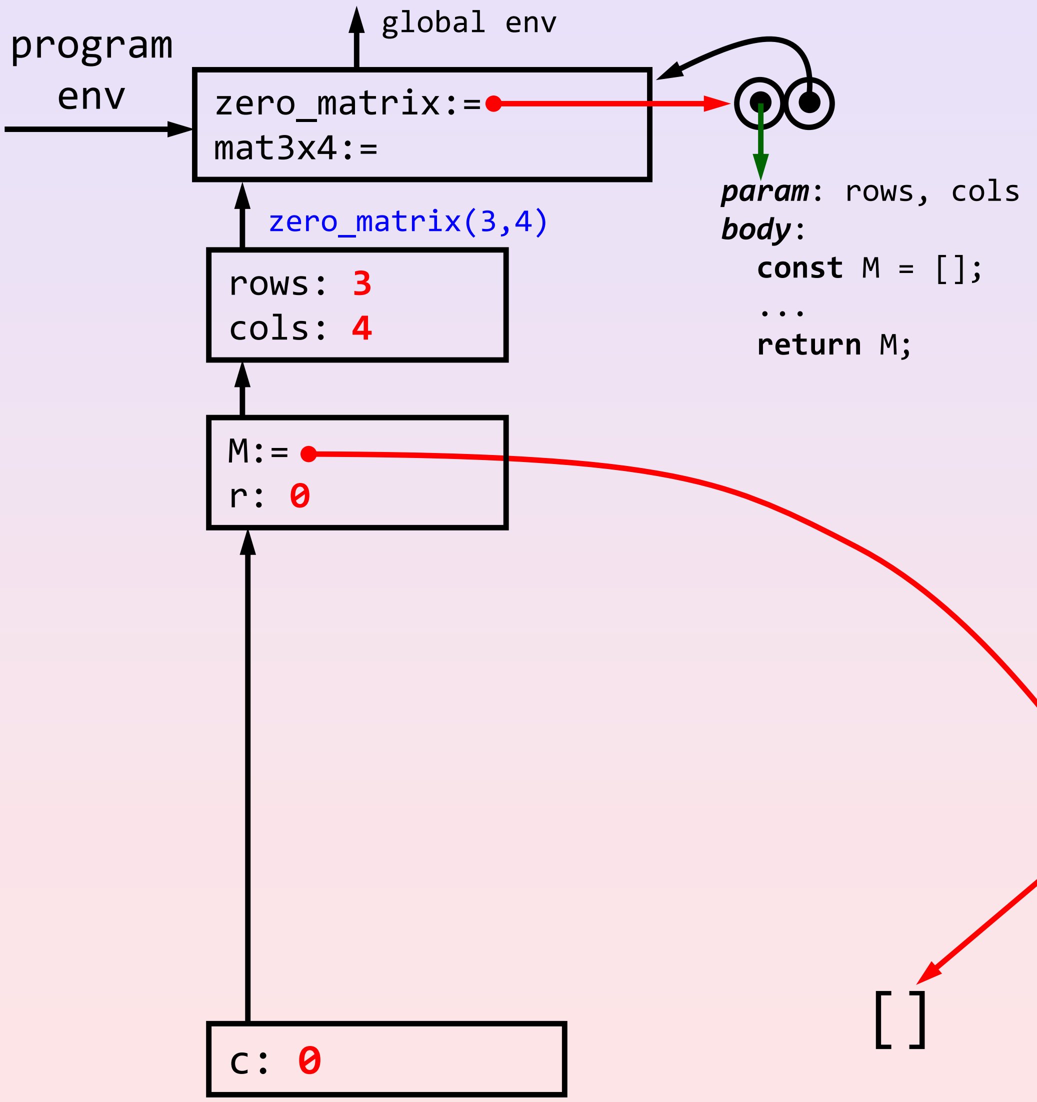
param: rows, cols
body:

```
   const M = [];
   ...
   return M;
```

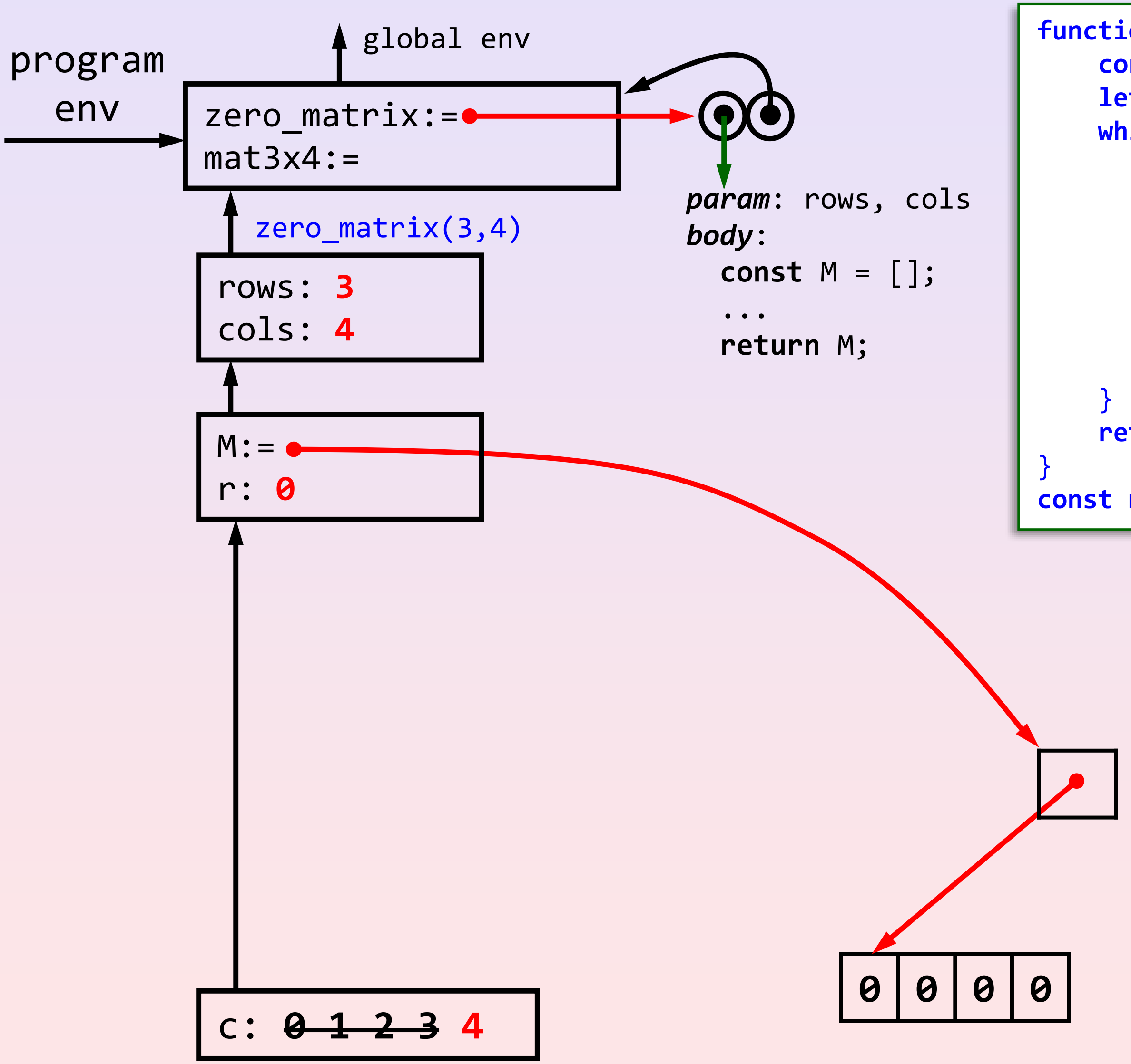
```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



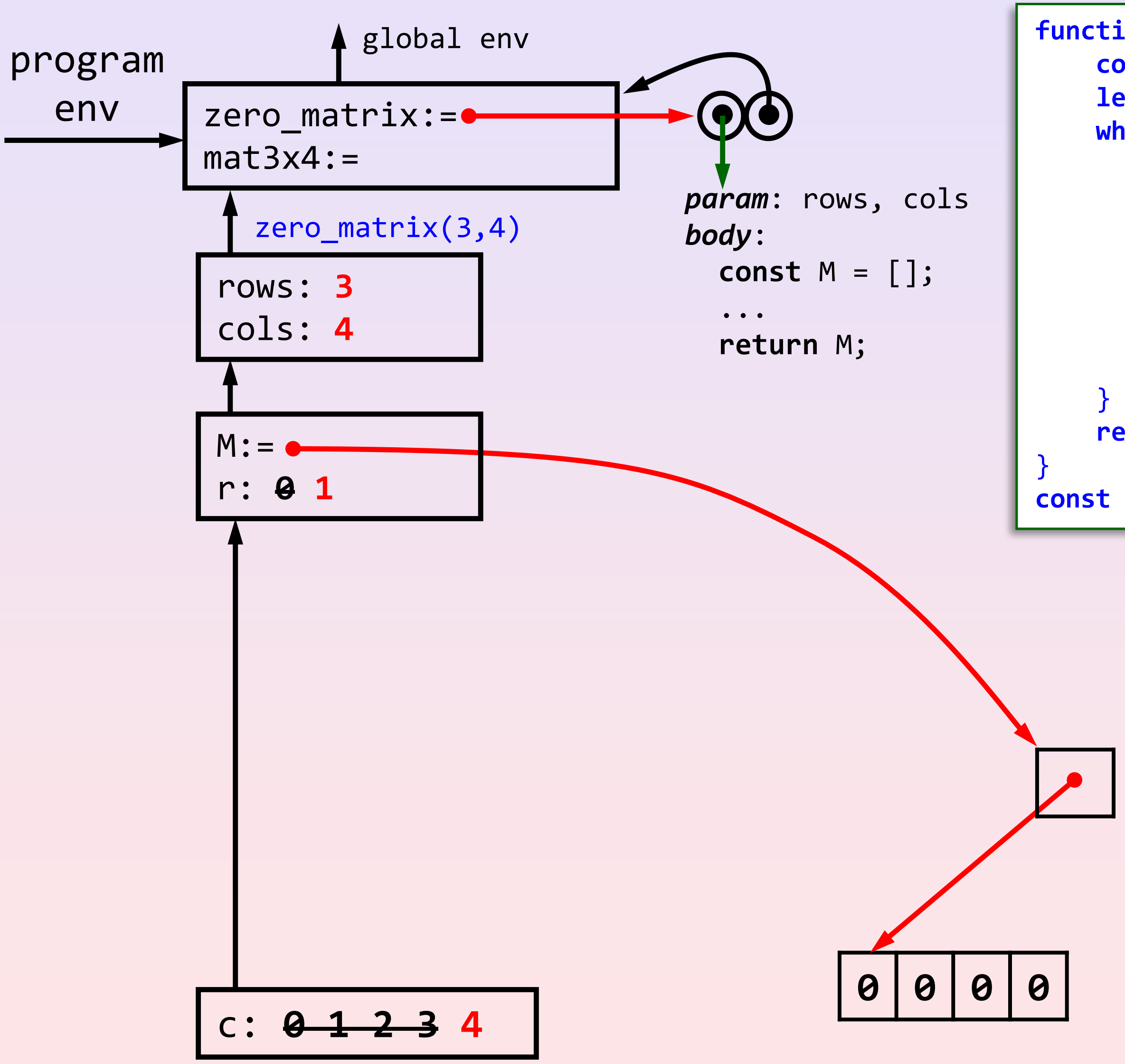
```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



program env

global env

zero_matrix:=
mat3x4:=

zero_matrix(3,4)

rows: 3
cols: 4

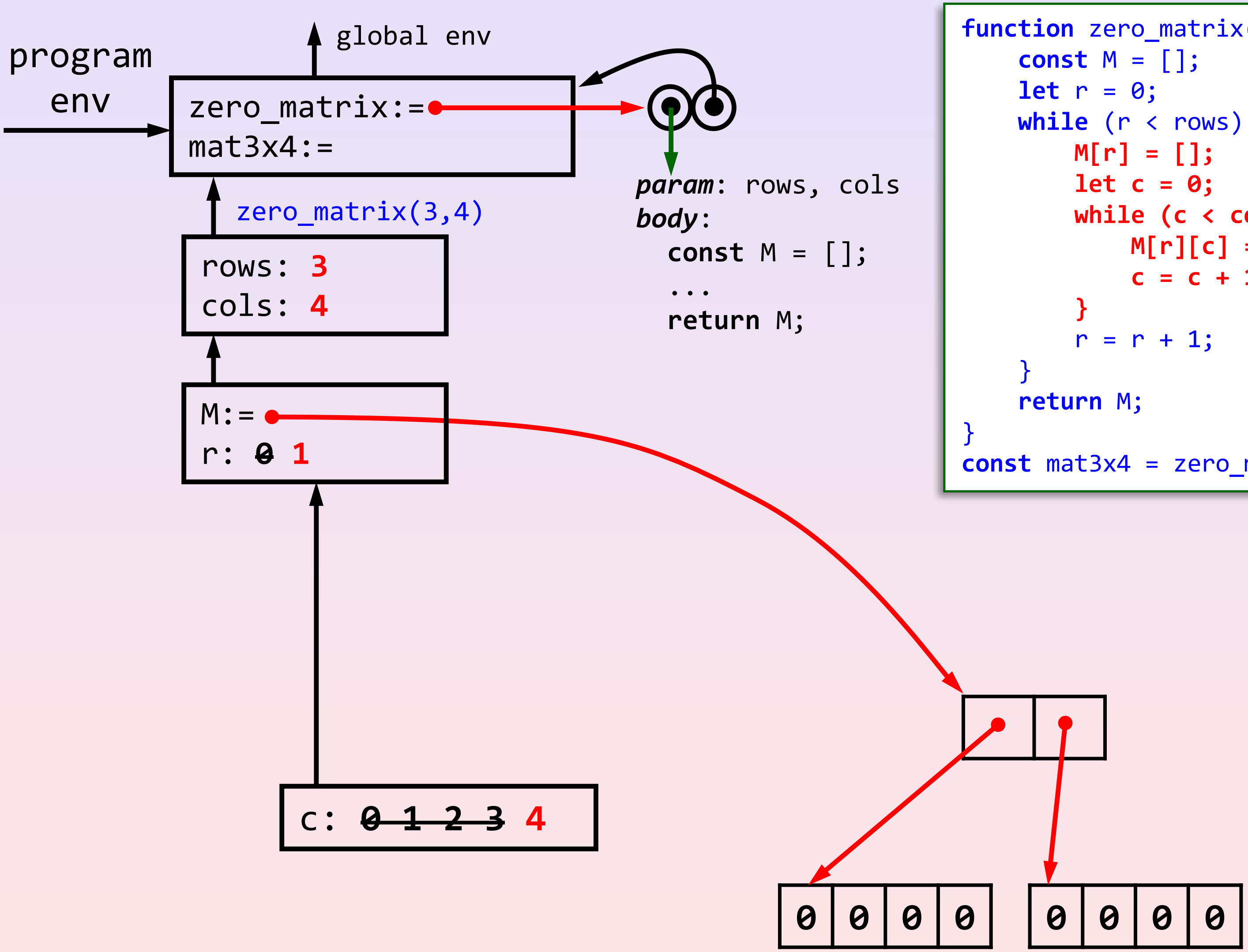
M:=
r: ~~0~~ 1

c: ~~0 1 2 3~~ 4

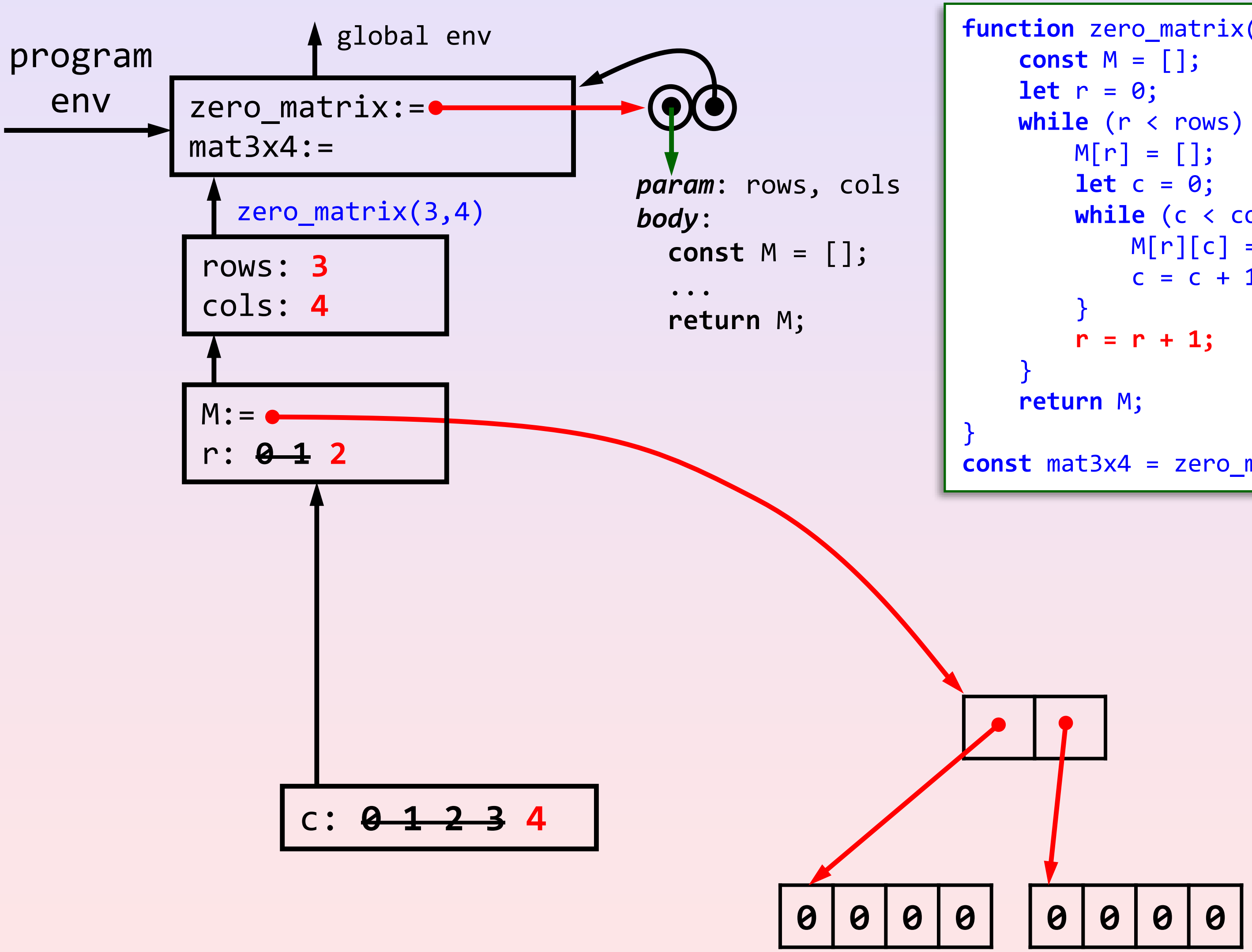
0 0 0 0

param: rows, cols
body:
```
const M = [];
...
return M;
```

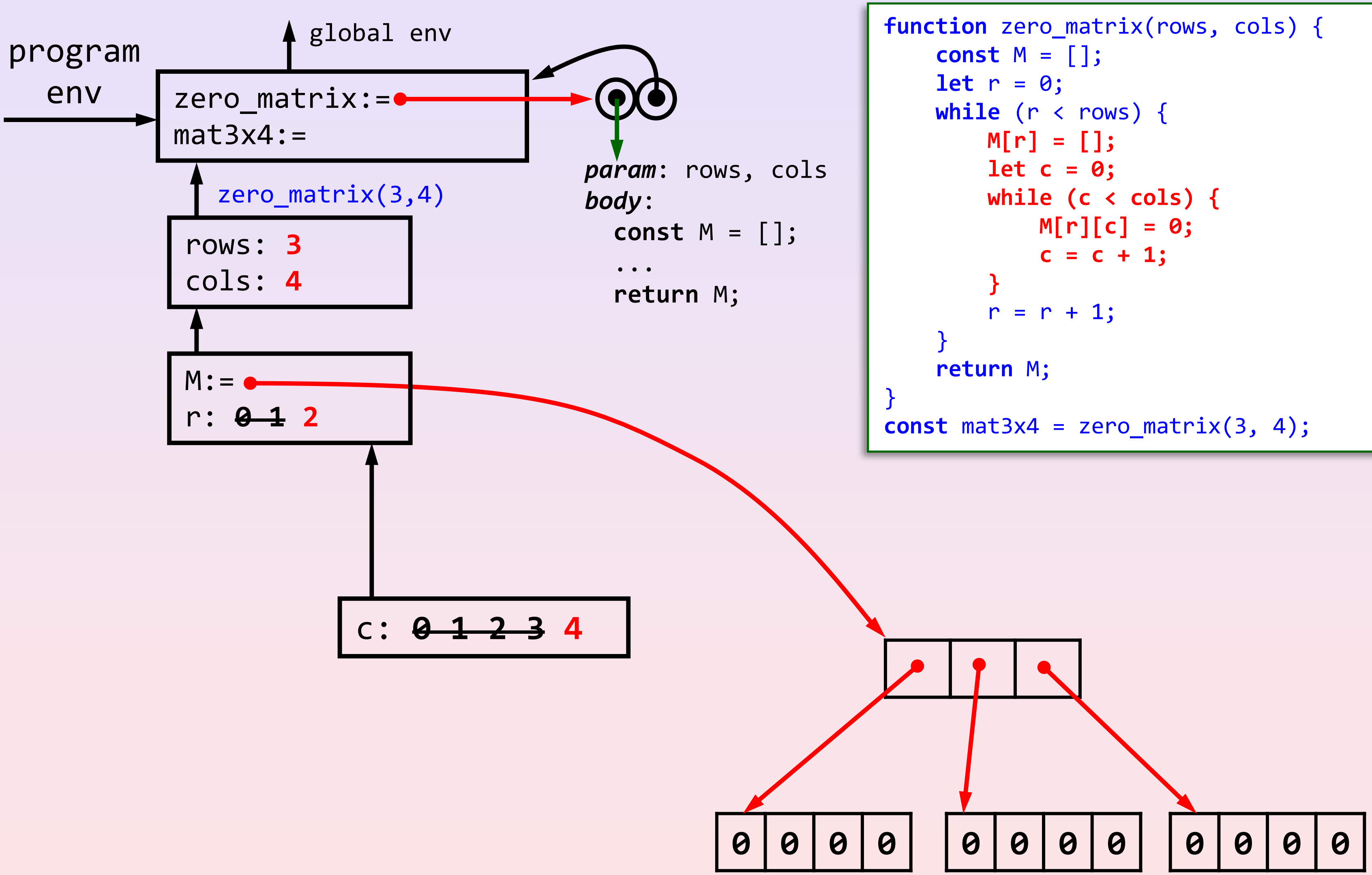
```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



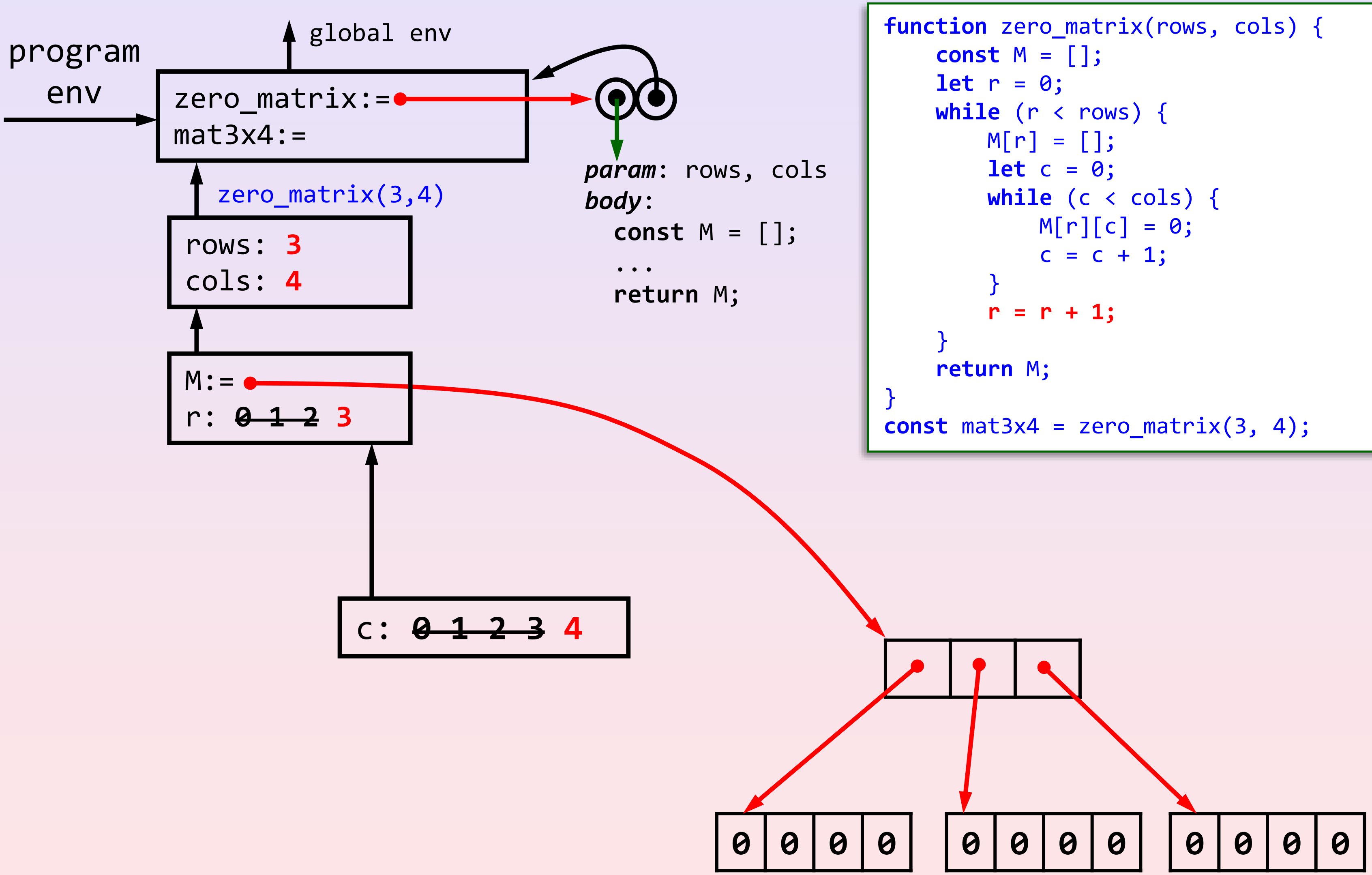
```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



program env

global env

zero_matrix:=

mat3x4:=

zero_matrix(3,4)

rows: 3

cols: 4

M:=

r: ~~0 1 2~~ 3

c: ~~0 1 2 3~~ 4

param: rows, cols

body:

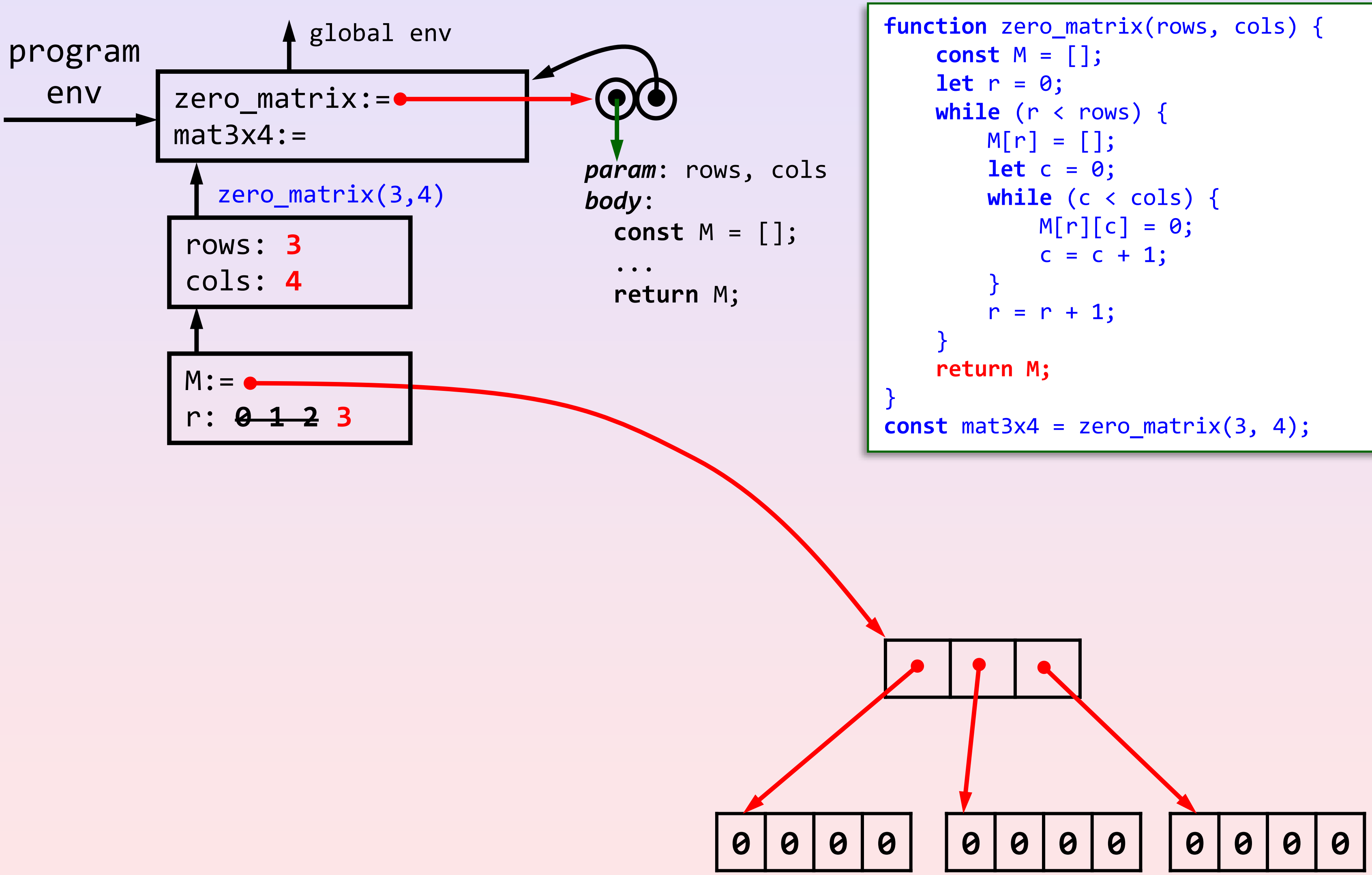
```
const M = [];
...
return M;
```

0	0	0	0

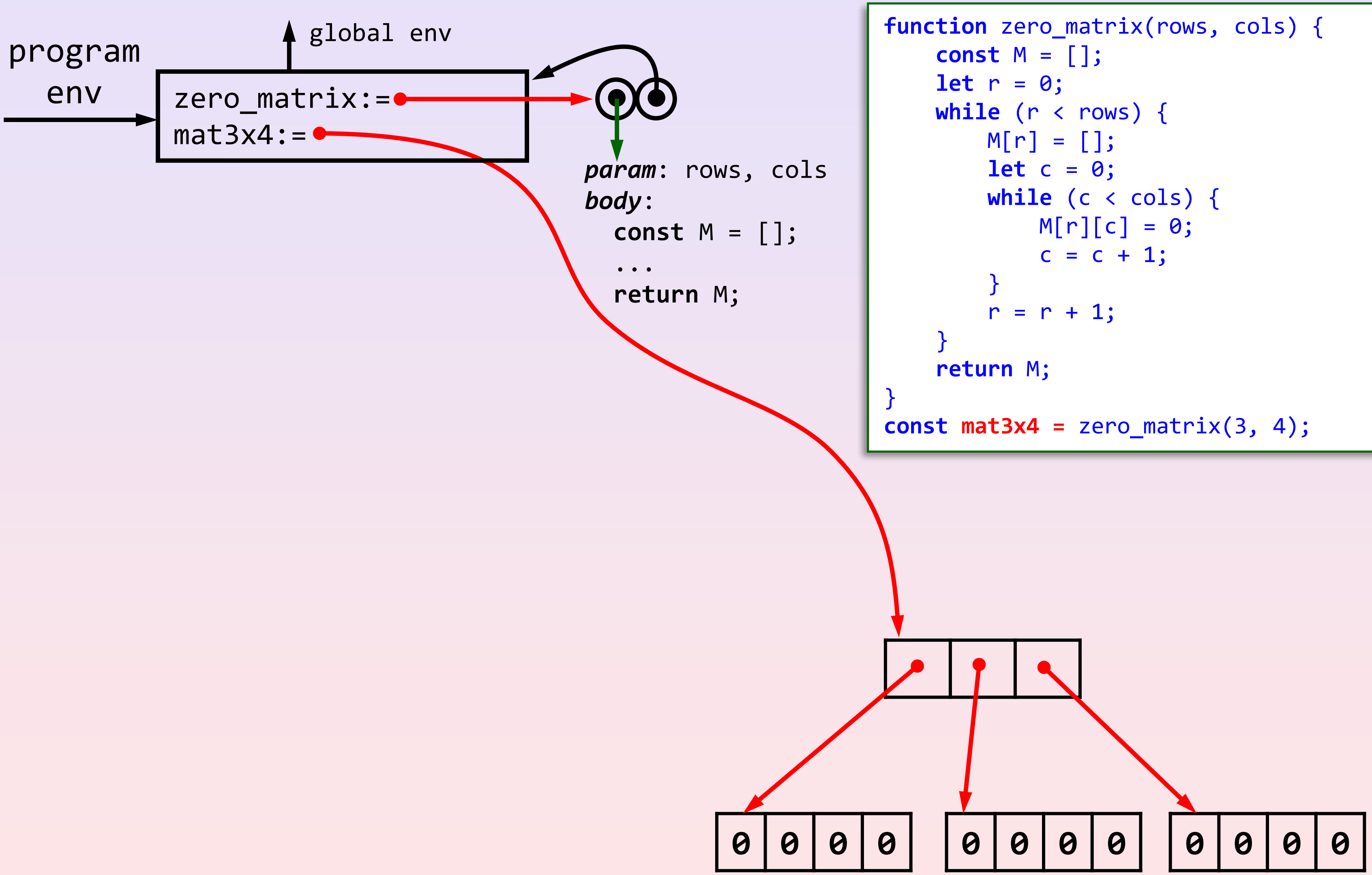
0	0	0	0

0	0	0	0

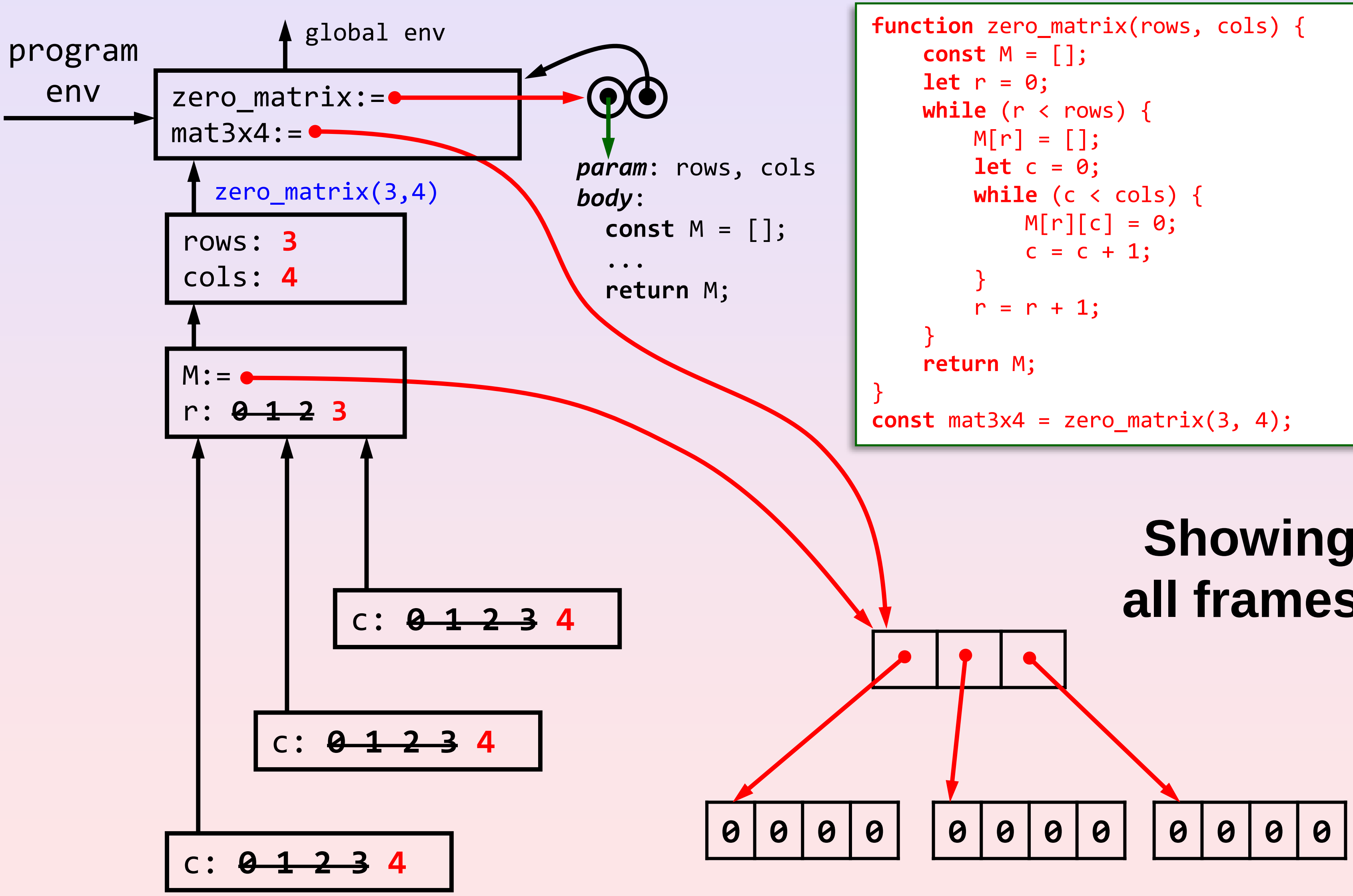
```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```



```
function zero_matrix(rows, cols) {
    const M = [];
    let r = 0;
    while (r < rows) {
        M[r] = [];
        let c = 0;
        while (c < cols) {
            M[r][c] = 0;
            c = c + 1;
        }
        r = r + 1;
    }
    return M;
}
const mat3x4 = zero_matrix(3, 4);
```

Showing all frames!

Order of Growth in Time of zero_matrix

```
function zero_matrix(rows, cols) {
    const M = [];
    for (let r = 0; r < rows; r = r + 1) {
        M[r] = [];
        for (let c = 0; c < cols; c = c + 1) {
            M[r][c] = 0;
        }
    }
    return M;
}
```

Show in Playground

- What is the order of growth in time?
  - Θ(rows * cols)

Summary

- **Arrays** support **random access** to the elements
- **Loops** are convenient for **iterative** computations
- **`for`** loops add convenience and readability to **`while`** loops
- **`break`** and **`continue`** add flexibility
- **Loops** can be **nested** inside other loops