SUDOKU Solution Validator

CMPE 180-94 Operating System Design
Spring 2017

Submitted by:

Team 19

Avni Gulati	011498227	avni.gulati@sjsu.edu
Rucha Apte	011824371	rucha.apte@sjsu.edu
Sowmya Gowrishankar	005699733	sowmya.gowrishankar@sjsu.edu

Course Instructor:
Dr. Hungwen Li

Table of Contents

Introduction

Sudoku is a logic-based number-placement puzzle game. The standard Sudoku puzzle is a table made up of 9 rows, 9 columns and 9, 3x3 boxes. The puzzle starts with given numbers in various positions and the player’s goal is to complete the table such that each row, column and box contains every number from 1 to 9 exactly once. Although the 9x9 variation of Sudoku is most common, larger variations exist to increase the puzzles difficulty. An example is a 16x16 variation made up of 16 rows, 16 columns and 16 4x4 boxes

Objective

- The objective of this project is to design a multi-threaded application in C language to determine whether the solution to a Sudoku puzzle is valid.
- A comparative study of a Sudoku solution validator using only one thread should also be done to understand if multi-threading approach improves performance for all programs.

Approach and Methodology

Creating threads to check each of the column, row and 3 x 3 sub-grids

To achieve this, we plan to create 27 threads in total

- 9 threads to check that each column contains the digits 1 through 9
- 9 threads to check that each row contains the digits 1 through 9
- 9 threads to check that each of the 3×3 sub-grids contains the digits 1 through 9

Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid.

Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent.

Functions to check the sudoku solution without using threads will also be included in this program.

The elapsed time for each of the approaches will also be displayed to understand the performance improvement/degradation.

Algorithm Design and Implementation

PThreads Approach

There are basic operations that can be reused. Every one of the validity checks (rows, columns, regions) is about making sure that all the digits from 1 through 9 are present exactly once in a certain set of 9 elements.

Step 1) Create the 27 threads. Create one thread each for the 9 3x3 grids, one thread each for the 9 columns and one thread each for the 9 rows. There will be a total of 27 thread created. The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid.

Step 2) Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent.

Step 3) Create an array of integer values called result[] that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise.

Step 4) Create a method to check each row if it contains all digits 1-9. There is an array called validarray[10] initialized to 0. For every value in the row, the corresponding index in validarray[] is checked for 0 and set to 1.

If the value in validarray[] is already 1, then it means that the value is repeating. So, the solution is invalid.

Step 5) Create a method to check each column if it contains all digits 1-9. There is an array called validarray[10] initialized to 0. For every value in the row, the corresponding index in validarray[] is checked for 0 and set to 1.

If the value in validarray[] is already 1, then it means that the value is repeating. So, the solution is invalid.

Step 6) Create a method to check each if a square of size 3x3 contains all numbers from 1-9. There is an array called validarray[10] initialized to 0. For every value in the square, the corresponding index in validarray[] is checked for 0 and set to 1.

If the value in validarray[] is already 1, then it means that the value is repeating. So, the solution is invalid.

Step 7) Wait for all threads to finish their tasks.

Step 8) When all worker threads have completed, the parent thread checks each entry in the result[] array to determine if the Sudoku puzzle is valid.

If the matrix passes every check then it is a valid matrix.

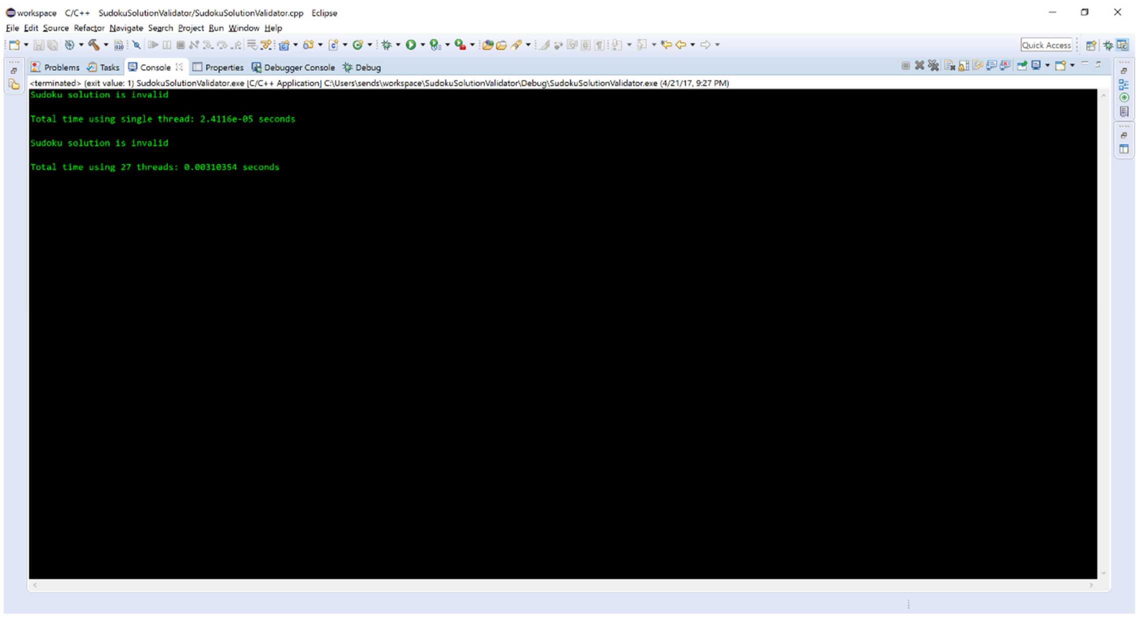
Single thread version
The row, column and grid check functions will be implemented and called without creating threads.

Test Results

Test Case 1:

Invalid Sudoku solution

```
{1, 2, 4, 5, 3, 9, 1, 8, 7},
{5, 1, 9, 7, 2, 8, 6, 3, 4},
{8, 3, 7, 6, 1, 4, 2, 9, 5},
{1, 4, 3, 8, 6, 5, 7, 2, 9},
{9, 5, 8, 2, 4, 7, 3, 6, 1},
{7, 6, 2, 3, 9, 1, 4, 5, 8},
{3, 7, 1, 9, 5, 6, 8, 4, 2},
{4, 9, 6, 1, 8, 2, 5, 7, 3},
{2, 8, 5, 4, 7, 3, 9, 1, 6}
```


workspace C/C++ SudokuSolutionValidator/SudokuSolutionValidator.cpp Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Tasks Console Properties Debugger Console Debug

```
<terminated> (exit value: 1) SudokuSolutionValidator.exe [C/C++ Application] C:\Users\sends\workspace\SudokuSolutionValidator\Debug\SudokuSolutionValidator.exe (4/21/17, 9:27 PM)
Sudoku solution is invalid

Total time using single thread: 2.4116e-05 seconds

Sudoku solution is invalid

Total time using 27 threads: 0.00310354 seconds
```

<u>Result:</u>

Sudoku solution is invalid

Total time using single thread: 2.3652e-05 seconds

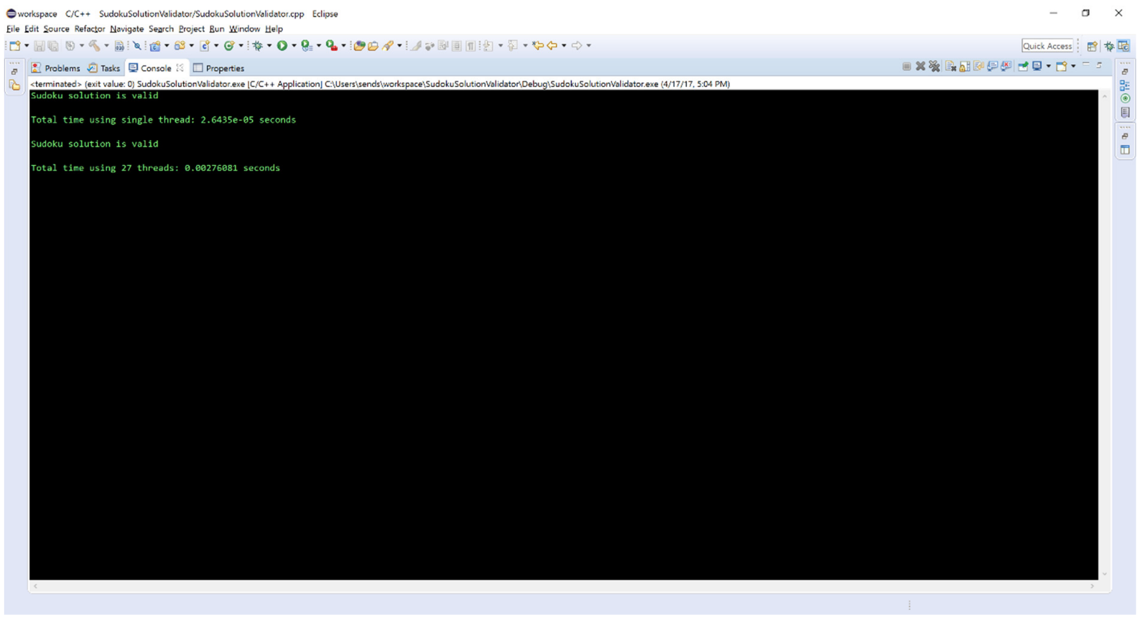
Sudoku solution is invalid

Total time: 0.00306968 seconds

Test Case 2:

Valid Sudoku solution

```
{6, 2, 4, 5, 3, 9, 1, 8, 7},
{5, 1, 9, 7, 2, 8, 6, 3, 4},
{8, 3, 7, 6, 1, 4, 2, 9, 5},
{1, 4, 3, 8, 6, 5, 7, 2, 9},
{9, 5, 8, 2, 4, 7, 3, 6, 1},
{7, 6, 2, 3, 9, 1, 4, 5, 8},
{3, 7, 1, 9, 5, 6, 8, 4, 2},
{4, 9, 6, 1, 8, 2, 5, 7, 3},
{2, 8, 5, 4, 7, 3, 9, 1, 6}
```


workspace C/C++ SudokuSolutionValidator/SudokuSolutionValidator.cpp Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Tasks Console Properties
Quick Access

```
<terminated> (exit value: 0) SudokuSolutionValidator.exe [C/C++ Application] C:\Users\sends\workspace\SudokuSolutionValidator\Debug\SudokuSolutionValidator.exe (4/17/17, 5:04 PM)
Sudoku solution is valid

Total time using single thread: 2.6435e-05 seconds

Sudoku solution is valid

Total time using 27 threads: 0.00276081 seconds
```

<u>Result:</u>

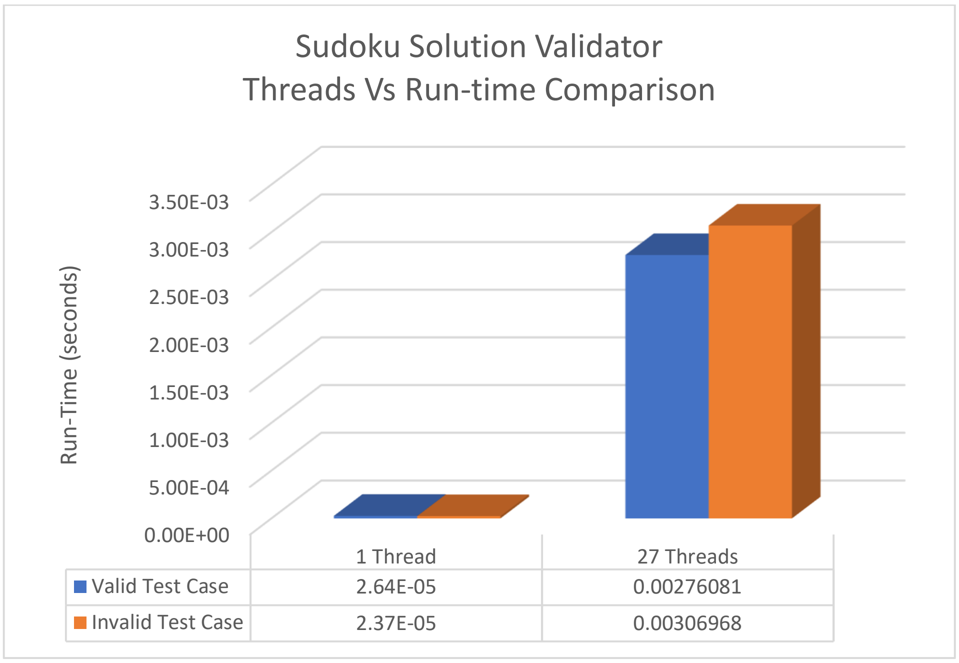
Sudoku solution is valid

Total time using single thread: 2.6435e-05 seconds

Sudoku solution is valid

Total time using 27 threads: 0.00276081 seconds

Performance Analysis



	1 Thread	27 Threads
🟦 Valid Test Case	2.64E-05	0.00276081
🟧 Invalid Test Case	2.37E-05	0.00306968

Conclusion

- For simple, short programs like Hexa-Sudoku, single thread faster than multi thread
- Due to thread creation & thread handling overhead, multithread is slower for such programs
- Multi-threading will definitely have advantages and performance improvement over Single thread for complex codes

Appendix: Code

```
//File Name: SudokuSolutionValidator.cpp
/*
 Authors:
    Avni Gulati         011498227
    Rucha Apte          011824371
    Sowmya Gowrishankar 005699733
 */
//Project Name: Sudoku Solution Validator using both Single thread and PThreads

//Description:
/*
 * This program takes a Sudoku puzzle solution as an input and then determines whether
 * the puzzle solution is valid. This validation is done using both single thread and 27
threads.
 * 27 threads are created as follows:
 * 9 for each 3x3 subsection, 9 for the 9 columns, and 9 for the 9 rows.
 * Each thread returns a integer value of 1 indicating that
 * the corresponding region in the puzzle they were responsible for is valid.
 * The program then waits for all threads to complete their execution and
 * checks if the return values of all the threads have been set to 1.
 * If yes, the solution is valid. If not, solution is invalid.
 *
 * This program also displays the total time taken for validation.
 */
//Last Changed: Mar 22, 2017

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <iostream>
#include <chrono>

#define num_threads 27

using namespace std;
using namespace std::chrono;
/**
 * Structure that holds the parameters passed to a thread.
 * This specifies where the thread should start verifying.
 */
typedef struct
{
    // The starting row.
    int row;
    // The starting column.
    int col;
    // The pointer to the sudoku puzzle.
    int (* board)[9];

} parameters;

/*
    Initialize the array which worker threads can update to 1 if the
    corresponding region of the sudoku puzzle they were responsible
    for is valid.
*/
int result[num_threads] = {0};

// Prototype for 3x3 square function.
void *check_grid(void *params);

// Prototype for the check_rows function.
```

```
void *check_rows(void *params);

// Prototype for the check_cols function.
void *check_cols(void *params);

// Prototype for the single thread sudoku check function.
int sudoku_checker(int sudoku[9][9]);

/***************
 * ENTRY POINT *
 **************/
int main(void)
{
    //====== The Sudoku puzzle to be validated =======
    int sudoku[9][9] =
    {
            {6, 2, 4, 5, 3, 9, 1, 8, 7},
            {5, 1, 9, 7, 2, 8, 6, 3, 4},
            {8, 3, 7, 6, 1, 4, 2, 9, 5},
            {1, 4, 3, 8, 6, 5, 7, 2, 9},
            {9, 5, 8, 2, 4, 7, 3, 6, 1},
            {7, 6, 2, 3, 9, 1, 4, 5, 8},
            {3, 7, 1, 9, 5, 6, 8, 4, 2},
            {4, 9, 6, 1, 8, 2, 5, 7, 3},
            {2, 8, 5, 4, 7, 3, 9, 1, 6}
     };

    // Starting time for single thread execution
    steady_clock::time_point start_time_single_thread = steady_clock::now();

    if(sudoku_checker(sudoku))
        printf("Sudoku solution is invalid\n");
    else
        printf("Sudoku solution is valid\n");

    // Compute and return the elapsed time in milliseconds.
    steady_clock::time_point end_time_single_thread = steady_clock::now();
    duration<double> elapsed_time_single_thread =
duration_cast<duration<double>>(end_time_single_thread - start_time_single_thread);

    cout << endl << "Total time using single thread: " <<
elapsed_time_single_thread.count() << " seconds" << endl << endl;

    // Starting time for execution with 27 threads
    steady_clock::time_point start_time_threads = steady_clock::now();

    pthread_t threads[num_threads];

    int threadIndex = 0;

    // ====== Create the threads ======
    //Create one thread each for the 9 3x3 grid, one thread each for the 9 columns and one
thread each for the 9 rows. There will be a total of 27 thread created
    //Function call parameters are Thread identifier, thread attributes, function the
thread executes, parameters passed to function
    //Syntax for pthread_create function is pthread_create(pthread_t * thread,
pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            // ====== Declaration of the parameter for the 3X3 grid check threads =======
            if (i%3 == 0 && j%3 == 0)
            {
                parameters *gridData = (parameters *) malloc(sizeof(parameters));
                gridData->row = i;
                gridData->col = j;
                gridData->board = sudoku;
                pthread_create(&threads[threadIndex++], NULL, check_grid, gridData);
            }

            // ====== Declaration of the parameter for the row check threads =======
            if (j == 0)
            {
                parameters *rowData = (parameters *) malloc(sizeof(parameters));
                rowData->row = i;
                rowData->col = j;
                rowData->board = sudoku;
                pthread_create(&threads[threadIndex++], NULL, check_rows, rowData);
            }

            // ====== Declaration of the parameter for the column check threads =======
            if (i == 0)
            {
                parameters *columnData = (parameters *) malloc(sizeof(parameters));
                columnData->row = i;
                columnData->col = j;
                columnData->board = sudoku;
                pthread_create(&threads[threadIndex++], NULL, check_cols, columnData);
            }
        }
    }

    // ======= Wait for all threads to finish their tasks =======
    //Parameters are Thread identifier and the return value of the function executed by the
thread
    for (int i = 0; i < num_threads; i++)
        pthread_join(threads[i], NULL);


    // If any of the entries in the valid array are 0, then the Sudoku solution is invalid
    for (int i = 0; i < num_threads; i++)
    {
        if (result[i] == 0)
        {
            cout << "Sudoku solution is invalid" << endl;

            // Compute and return the elapsed time in milliseconds.
            steady_clock::time_point end_time_threads = steady_clock::now();
            duration<double> elapsed_time_threads =
duration_cast<duration<double>>(end_time_threads - start_time_threads);

            cout << endl << " Total time using 27 threads: " <<
elapsed_time_threads.count() << " seconds" << endl;
            return 1;
        }
    }
    cout << "Sudoku solution is valid" << endl;

    // Compute and return the elapsed time in milliseconds.
    steady_clock::time_point end_time_threads = steady_clock::now();
    duration<double> elapsed_time_threads =
duration_cast<duration<double>>(end_time_threads - start_time_threads);

    cout << endl << "Total time using 27 threads: " << elapsed_time_threads.count() << "
seconds" << endl;

}

/*
 * Checks if a square of size 3x3 contains all numbers from 1-9.
 * There is an array called validarray[10] initialized to 0.
 * For every value in the square, the corresponding index in validarray[] is checked for 0
and set to 1.
 * If the value in validarray[] is already 1, then it means that the value is repeating.
So, the solution is invalid.
 *
 * @param   void *      The parameters (pointer).
 */
void *check_grid(void * params)
{
    parameters *data = (parameters *) params;
    int startRow = data->row;
    int startCol = data->col;
    int validarray[10] = {0};
    for (int i = startRow; i < startRow + 3; ++i)
    {
        for (int j = startCol; j < startCol + 3; ++j)
        {
            int val = data->board[i][j];
            if (validarray[val] != 0)
                pthread_exit(NULL);
            else
                validarray[val] = 1;
        }
    }

    // If the execution has reached this point, then the 3x3 sub-grid is valid.
    result[startRow + startCol/3] = 1; // Maps the 3X3 sub-grid to an index in the first 9
indices of the result array
    pthread_exit(NULL);
}

/**
 * Checks each row if it contains all digits 1-9.
 * There is an array called validarray[10] initialized to 0.
 * For every value in the row, the corresponding index in validarray[] is checked for 0 and
set to 1.
 * If the value in validarray[] is already 1, then it means that the value is repeating.
So, the solution is invalid.
 *
 * @param   void *      The parameters (pointer).
 */

void *check_rows(void *params)
{
    parameters *data = (parameters *) params;
    int row = data->row;

    int validarray[10] = {0};
    for (int j = 0; j < 9; j++)
    {
        int val = data->board[row][j];
        if (validarray[val] != 0)
            pthread_exit(NULL);
        else
            validarray[val] = 1;
    }

    // If the execution has reached this point, then the row is valid.
    result[9 + row] = 1; // Maps the row to an index in the second set of 9 indices of the
result array
    pthread_exit(NULL);
}

/**
 * Checks each column if it contains all digits 1-9.
 * There is an array called validarray[10] initialized to 0.
 * For every value in the row, the corresponding index in validarray[] is checked for 0 and
set to 1.
 * If the value in validarray[] is already 1, then it means that the value is repeating.
So, the solution is invalid.
 *
 * @param   void *      The parameters (pointer).
 */
void *check_cols(void *params)
{
    parameters *data = (parameters *) params;
    //int startRow = data->row;
    int col = data->col;

    int validarray[10] = {0};
    for (int i = 0; i < 9; i++)
    {
        int val = data->board[i][col];
        if (validarray[val] != 0)
            pthread_exit(NULL);
        else
            validarray[val] = 1;
    }

    // If the execution has reached this point, then the column is valid.
    result[18 + col] = 1; // Maps the column to an index in the third set of 9 indices of
the result array
    pthread_exit(NULL);
 }

/**
 * Checks each column/row if it contains all digits 1-9.
 * There is an array called validarray[10] initialized to 0.
 * For every value in the row/column, the corresponding index in validarray[] is checked
for 0 and set to 1.
 * If the value in validarray[] is already 1, then it means that the value is repeating.
So, the solution is invalid.
 *
 * @param   int      the row/column to be checked.
 */
int check_line(int input[9])
{
    int validarray[10] = {0};
    for (int i = 0; i < 9; i++)
    {
        int val = input[i];
        if (validarray[val] != 0)
            return 1;
        else
            validarray[val] = 1;
    }
    return 0;
}

/**
 * Checks each 3*3 grid if it contains all digits 1-9.
 * There is an array called validarray[10] initialized to 0.
 * For every value in the row/column, the corresponding index in validarray[] is checked
for 0 and set to 1.
 * If the value in validarray[] is already 1, then it means that the value is repeating.
So, the solution is invalid.
 *
 * @param   void *      The parameters (pointer).
 */
int check_grid(int sudoku[9][9])
{
    int temp_row, temp_col;
```

```c
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            temp_row = 3 * i;
            temp_col = 3 * j;
            int validarray[10] = {0};

            for(int p=temp_row; p < temp_row+3; p++)
            {
                for(int q=temp_col; q < temp_col+3; q++)
                {
                    int val = sudoku[p][q];
                    if (validarray[val] != 0)
                        return 1;
                    else
                        validarray[val] = 1;
                }
            }
        }
    }
    return 0;
}

/**
 * Checks if the sudoku solution is valid or not without using the PThreads function.
 *
 *
 * @param   int      The sudoku solution.
 */
int sudoku_checker(int sudoku[9][9])
{
    for (int i=0; i<9; i++)
    {
        /* check row */
        if(check_line(sudoku[i]))
            return 1;

        int check_col[9];
        for (int j=0; j<9; j++)
            check_col[j] = sudoku[i][j];

        /* check column */
        if(check_line(check_col))
            return 1;

        /* check grid */
        if(check_grid(sudoku))
            return 1;
    }
    return 0;
}
```