*"Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity. ... The geniuses of the computer field, on the the other hand, are the people with the keenest aesthetic senses, the ones who are capable of creating beauty. Beauty is decisive at every level: the most important interfaces, the most important programming languages, the winning algorithms are the beautiful ones."*

D. Gelernter ("Machine Beauty", Basic Books, 1998)

*"The road to wisdom? Well, it's plain and simple to express: Err and err and err again, but less and less and less."* Piet Hein

# This is a solo lab!

# Learning Objectives

1. More practice writing MIPS code with conditionals, array refs, and calling conventions
2. To use function calls and recursion in MIPS assembly

# Work that needs to be handed in (via github)

- `p1.s`: implement the `value` function. **Run on EWS with:**
  `QtSpim -file p1_main.s p1.s`
- `p2.s`: implement the `encode_domino` and `solve` functions. **Run on EWS with:**
  `QtSpim -file p2_main.s p2.s -limit 20000000`



# Additional Information

For Lab 7 we are providing "main" files (*e.g.*, `p1_main.s`) and files for you to implement your functions (*e.g.*, `p1.s`), just like in Lab 7. We will need to load both of these files into QtSpim to test your code.

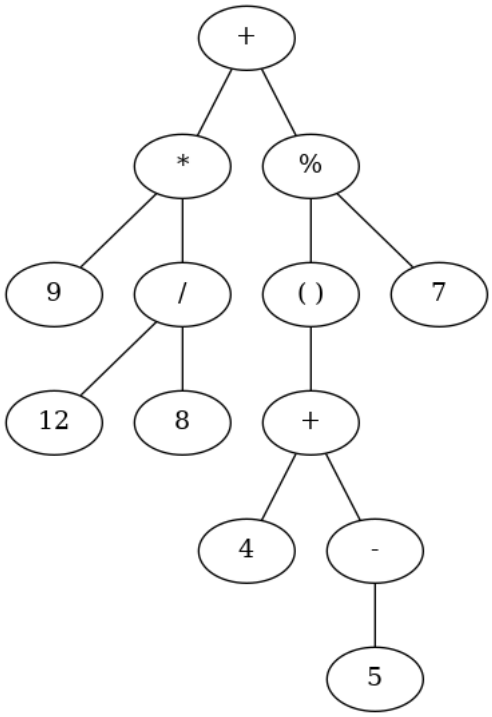
We will only be grading the `p1.s` and `p2.s` files, and we will do so with our own copy of `p*_main.s`, so make sure that your code works correctly with an original copy of `p*_main.s`. We provide you with a `code.c` file that contains C implementations of all the functions that you need to implement for this lab.

# Guidelines

- Guidelines are the same as Lab 7.
- You'll find the assignment *so much easier* if you try to understand the C code you're translating before starting.
- Just as in Lab 6, follow all calling and register-saving conventions you've learned. It's even more important in this lab.
- Don't try to change the algorithms at this point, just write the code in MIPS as closely to the provided C code as possible.
- **Remember to test your code using the test cases provided!**

# Problem 1: AST Evaluation [20 points]

In this problem, your job is to write an assembly program to traverse an abstract syntax tree (AST) and evaluate it. An AST represents some construct in whatever language it is representing. Here, we will be using it to represent a subset of algebraic expressions. An example AST that priorities division over multiplication might look like:




9 * 12 / 8 + (4 + -5) % 7

This task is a fairly simple tree traversal. For this function, we will have you support the additive (+, -), multiplicative operations (*, /, %), negation(-), and parenthetical operations. These should be straight forward, you may implement the switch case in any manner that is valid, but it might make your code cleaner to have an array of function addresses and call the functions based on the value of node type. In addition to evaluating the AST, we will also be memoizing the results. This is important because **we will reuse and modify the left and right pointers of the AST nodes**. This means that failing test cases can have side effects.

It might be helpful to note that pseudoinstructions similar to add and sub exist that make some of these operations simple. There are an abstraction over how MIPS does these operations with the $HI and $LO registers.

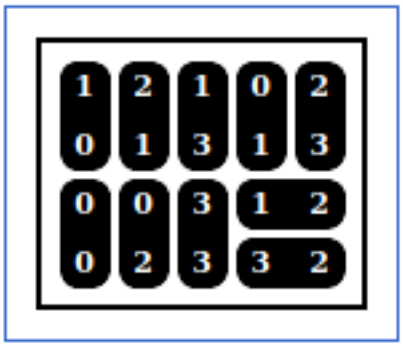
```
div $t0, $t1, $t2 # Computes division
mul $t0, $t1, $t2 # Computes multiplication
rem $t0, $t1, $t2 # Computes remainder

div $t0, $t1, 2 # Computes division
mul $t0, $t1, 2 # Computes multiplication
rem $t0, $t1, 2 # Computes remainder
```

```
#define NULL 0

// Note that the value of op_add is 0 and the value of each item
// increments as you go down the list
//
// In C, an enum is just an int!
typedef enum {
    op_add,
    op_sub,
    op_mul,
    op_div,
    op_rem,
    op_neg,
    op_paren,
    constant
} node_type_t;

typedef struct {
    node_type_t type;
    bool computed;
    int value;
    ast_node* left;
    ast_node* right;
} ast_node;

int value(ast_node* node) {
    if (node == NULL) { return 0; }
    if (node->computed) { return node->value; }

    int left = value(node->left);
    int right = value(node->right);

    // This can just implemented with successive if statements (see Appendix)
    switch (node->type) {
        case constant:
            return node->value;
        case op_add:
            node->value = left + right;
            break;
        case op_sub:
            node->value = left - right;
            break;
        case op_mul:
            node->value = left * right;
            break;
        case op_div:
            node->value = left / right;
            break;
        case op_rem:
            node->value = left % right;
            break;
        case op_neg:
            node->value = -left;
            break;
        case op_paren:
            node->value = left;
            break;
    }
    node->computed = true;
```

```
    return node->value;
}
```

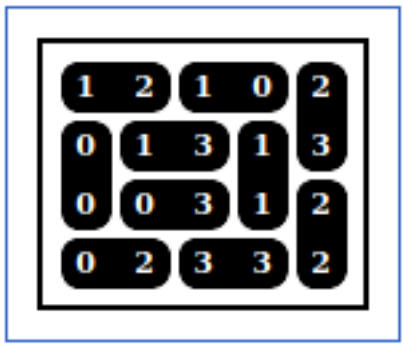
## Problem 2: Dominosa [80 points]

Dominosa is a game where you try to tile a board with dominoes while never reusing the same domino twice. You can play the game here `https://www.puzzle-dominosa.com/` to get some intuition about it. Therefore,




This is invalid since the domino $(1, 0)$ is reused twice

Whereas,




This is valid since every domino is unique

You need to implement `encode_domino` and `solve`. We've provided a sample game in `p2_main.s`. It does not cover all test cases we will use to grade your code and we encourage you to create more to test edge cases.

The following is an inefficient recursive descent algorithm that tries every combination of dominoes until it finds a configuration that both covers the board and uses unique tiles. There are three main issues that you may run into with this lab.

- The solution takes too long. On a slow machine, it may take up to 30 seconds to solve, so this is somewhat normal. However we only give you 20,000,000 cycles to solve the puzzle. You will not need to go over it.
- Running out of registers. If you run out of registers, break things into smaller functions. You can alternatively use the caller-saved pattern, but that is more error prone as you duplicate more code.
- Infinite loops or early returns. This typically happens because the stack was managed incorrectly and values got corrupted. Before asking a CA for help, double check that you haven't messed up your stack.

```
#define MAX_GRIDSIZE 16
#define MAX_MAXDOTS 15

// puzzle question structure
typedef struct {
    int num_rows;
    int num_cols;
    int max_dots;
    unsigned char board[MAX_GRIDSIZE * MAX_GRIDSIZE];
    unsigned char dominos_used[MAX_MAXDOTS * MAX_MAXDOTS];
} dominosa_question;

/*** begin of the solution to the puzzle ***/
// zero out an array with given number of elements
void zero(int num_elements, unsigned char* array) {
    for (int i = 0; i < num_elements; i++) {
        array[i] = 0;
    }
}
// encode each domino as an int
int encode_domino(unsigned char dots1, unsigned char dots2, int max_dots) {
    return dots1 < dots2 ? dots1 * max_dots + dots2 + 1 : dots2 * max_dots + dots1 + 1;
}

// main solve function, recurse using backtrack
// puzzle is the puzzle question struct
// solution is an array that the function will fill the answer in
// row, col are the current location
// dominos_used is a helper array of booleans (represented by a char)
//   that shows which dominos have been used at this stage of the search
//   use encode_domino() for indexing
int solve(dominosa_question* puzzle,
          unsigned char* solution,
          int row,
          int col) {

    int num_rows = puzzle->num_rows;
    int num_cols = puzzle->num_cols;
    int max_dots = puzzle->max_dots;
    int next_row = ((col == num_cols - 1) ? row + 1 : row);
    int next_col = (col + 1) % num_cols;
    unsigned char* dominos_used = puzzle->dominos_used;

    if (row >= num_rows || col >= num_cols) { return 1; }
    if (solution[row * num_cols + col] != 0) {
        return solve(puzzle, solution, next_row, next_col);
    }

    unsigned char curr_dots = puzzle->board[row * num_cols + col];

    if (row < num_rows - 1 && solution[(row + 1) * num_cols + col] == 0) {
        int domino_code = encode_domino(curr_dots,
                                        puzzle->board[(row + 1) * num_cols + col],
                                        max_dots);

        if (dominos_used[domino_code] == 0) {
            dominos_used[domino_code] = 1;
            solution[row * num_cols + col] = domino_code;
            solution[(row + 1) * num_cols + col] = domino_code;
```

```
            if (solve(puzzle, solution, next_row, next_col)) {
                return 1;
            }
            dominos_used[domino_code] = 0;
            solution[row * num_cols + col] = 0;
            solution[(row + 1) * num_cols + col] = 0;
        }
    }
    if (col < num_cols - 1 && solution[row * num_cols + (col + 1)] == 0) {
        int domino_code = encode_domino(curr_dots,
                                        puzzle->board[row * num_cols + (col + 1)],
                                        max_dots);
        if (dominos_used[domino_code] == 0) {
            dominos_used[domino_code] = 1;
            solution[row * num_cols + col] = domino_code;
            solution[row * num_cols + (col + 1)] = domino_code;
            if (solve(puzzle, solution, next_row, next_col)) {
                return 1;
            }
            dominos_used[domino_code] = 0;
            solution[row * num_cols + col] = 0;
            solution[row * num_cols + (col + 1)] = 0;
        }
    }
    return 0;
}

// the slow solve entry function,
// solution will appear in solution array
// return value shows if the dominosa is solved or not
int slow_solve_dominosa(dominosa_question* puzzle, unsigned char* solution) {
    zero(puzzle->num_rows * puzzle->num_cols, solution);
    unsigned char dominos_used[MAX_MAXDOTS * MAX_MAXDOTS];
    zero(MAX_MAXDOTS * MAX_MAXDOTS, dominos_used);
    return solve(puzzle, solution, 0, 0, dominos_used);
}
// end of solution
/*** end of the solution to the puzzle ***/
```

# Appendix

There are two main concepts that you may not be familiar with: enums and switch-case.

## Enums

In C, an enum is little more than a named int. They occupy the same space as an int and are typically used as a more readable alternative constants. By default, the first entry in the enum has the value 0 and every successive entry has an increasing value. So, the following are identical.

```
typedef enum {
    zero,
    one,
    two,
    three
} constant_t;

typedef enum {
    zero = 0,
    one = 1,
    two = 2,
    three = 3
} constant_t;
```

## Switch-Case

The other construct presented is a switch-case statement. A switch statement is made of an input expression and cases. These cases are points in the code that will get jumped to depending on the value of the expression. Quite strangely, after navigating to a case, your code will continue executing until it sees a break, a return, or the end of the switch block. Switch is commonly used to select behavior among distinct states. The compiler will often replace the switch case with a jump table, which has a performance advantage to if-else asymptotically, but that is outside of the scope of this class.



```
switch (expression) {

    case val: // fallthrough
        printf("Expression = val1");
    case val2:
        printf("Expression = val1 or Expression = val2");
        return;

    case val3:
        printf("Expression = val3");
        break;

    default:
        printf("Default Case");
}

// Equivalently

if (expression == val1 || expression == val2) {
    if (expression == val1) {
        printf("Expression = val1");
    }
```

```
    printf("Expression = val1 or Expression = val2");
    return;
} else if (expression == val3) {
    printf("Expression = val3");
} else {
    printf("Default Case");
}
```