```
CC=gcc                          # c compiler
CFLAGS=-c -Wall                  # compiler flags
LDFLAGS=                        # linker arguments
SOURCES=main.c tritone.c vec.c vecvec.c ast.c  # source files
OBJECTS=$(patsubst %.c,build/%.o,$(SOURCES))
DEPS=$(patsubst %.o,%.d,$(OBJECTS))
EXECUTABLE=build/tritone

all: $(EXECUTABLE)

# pull in dependency info for *existing* .o files
-include $(DEPS)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(OBJECTS) $(LDFLAGS) -o $@
    ./$@

build/%.o: %.c
    $(CC) $(CFLAGS) $< -o $@
    $(CC) -MM $< > build/$*.d

clean:
    rm -rf build/*.o build/*.d $(EXECUTABLE)
```

```c
/**
 * @file main.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief Tritone: a bad vector calculator
 * Supports
 *
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tritone.h"


int main(int arc, char** argv) {
    if(argv[1] && !strcmp("-h", argv[1])) {
        print_help();
        exit(0);
    }

    do {
        printf("%s", tritone());
    } while(1);

    return -1;
}
```

```
1  /**
2   * @file tritone.h
3   * @author Caleb Andreano (andreanoc@msoe.edu)
4   * @class CPE2600-121
5   * @brief Tritone: a bad vector calculator
6   *
7   * Course: CPE2600-121
8   * Assignment: Lab Wk 5
9   * @date 2023-10-01
10  */
11
12 #ifndef TRITONE_H
13 #define TRITONE_H
14
15     char* tritone(void);
16     void print_help();
17
18 #endif
```

```c
/**
 * @file tritone.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief Tritone: a bad vector calculator
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#include <stdio.h>
#include <string.h>
#include "tritone.h"
#include "ast.h"
#include "vec.h"
#include "vecvec.h"

/**
 * @brief Runs the tritone application and returns it's output string
 *
 * @return char*
 */
char* tritone(void) {

    static int started = 0;
    if(!started) {
        printf("\033[0;35m");
        printf(" ____ ____ ____ ____ _____ _ _ ____  |\\\n");
        printf("(_ _)( _ \\(_ _)(_ _)( _ )( \\( )( ___)  |/\n");
        printf(" )(  )  / _)(_   )(  )(_)(  )  (  )__)  /|\n");
        printf("(__) (_)\\_)(____) (__) (_____)(_)\\_)(____) (')\n");
        printf(" type 'help' for help                \"| \n");
        printf("\n\033[0m");
        started = 1;

    }
    static char input_buffer[300];
    static char output_buffer[300];

    printf("\033[0;35m");
    printf("tritone");
    printf("\033[0m");
    printf("> ");

    fgets(input_buffer, 300, stdin);
    node* root = parse_input(input_buffer);
    strncpy(output_buffer, value_to_string(evaluate_ast(root)), 300);

    free_ast(root);
    return output_buffer;
}


/**
 * @brief Prints the help text
 */
void print_help() {
    printf("tritone: very bad vector calculator\n"
           "- store a vector: a = 1, 2, 3\n"
           "- scalar operations: 1+2, 6-9, 5*3, 9/1,\n"
           "- vector operations: a + b, a + (1, 2, 3 * c)\n"
           "\t-supports addition, subtraction, scalar multiplication,"
           " scalar division, cross product, dot product.\n"
```

```
65                      " help: print this message\n"
66                      " clear: clear the screen\n"
67                      " free: free all variables\n"
68                      " list: list all variables\n"
69                      );
70  }
```

```c
#ifndef VEC_H
#define VEC_H

    typedef struct {
        float i;
        float j;
        float k;
    } vector;


    vector vec_add(vector a, vector b);
    vector vec_sub(vector a, vector b);
    vector vec_mul(vector a, vector b);
    float vec_dot(vector a, vector b);
    vector vec_cross(vector a, vector b);
    char* vector_to_string(vector v);
    vector vec_max(void);
    int is_max(vector a);
    int free_vector(char* name);

#endif
```

```c
/**
 * @file vec.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief 3-dimensional vector struct and related mathematical operations
 *
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#include "vec.h"
#include "vecvec.h"
#include <stdio.h>
#include <float.h>

/**
 * @brief Adds two vectors together and returns their sum
 *
 * @param a
 * @param b
 * @return vector
 */
vector vec_add(vector a, vector b) {
    vector sum = { a.i + b.i, a.j + b.j, a.k + b.k } ;
    return sum;
}

/**
 * @brief Subtracts two vectors and returns their difference
 *
 * @param a
 * @param b
 * @return vector
 */
vector vec_sub(vector a, vector b) {
    vector diff = { a.i - b.i, a.j - b.j, a.k - b.k } ;
    return diff;
}

/**
 * @brief Multiplies two vectors element-wise
 *
 * @param a
 * @param b
 * @return vector
 */
vector vec_mul(vector a, vector b) {
    vector prod = { a.i * b.i, a.j * b.j, a.k * b.k } ;
    return prod;
}

/**
 * @brief Takes the dot product of two vectors
 *
 * @param a
 * @param b
 * @return float
 */
float vec_dot(vector a, vector b) {
    return (a.i * b.i) + (a.j * b.j) + (a.k * b.k);
}

```

```c
/**
 * @brief Takes the cros product of two vectors
 *
 * @param a
 * @param b
 * @return vector
 */
vector vec_cross(vector a, vector b) {
    float i = (a.j * b.k)  - (a.k * b.j);
    float j = -((a.i * b.k)  - (a.k * b.i));
    float k = (a.i * b.j)  - (a.j * b.i);
    vector cross = {i, j, k};
    return cross;
}

/**
 * @brief Converts a vector to a formatted string
 *
 * @param v
 * @return char*
 */
char* vector_to_string(vector v) {
    static char buffer[60];
    snprintf(buffer, 60, "{ i: %.2f, j: %.2f, k: %.2f }", v.i, v.j, v.k);
    return buffer;
}
```

```h
#ifndef VECVEC_H
#define VECVEC_H

    #define MAX_VECS 10
    #include "vec.h"

    typedef enum {
        OPEN = 0,
        CLOSED,
    } vec_state;

    typedef struct {
        vec_state state;
        char name[12];
        vector vec;
    } vec_cell;

    int clear_vectors();
    int insert_vector(vector vec, char name[]);
    vec_cell* get_vector(char* name);
    char* vec_cell_to_string(vec_cell* v);
    void list_vectors();

#endif
```

```c
/**
 * @file vecvec.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief Array of vectors. Supports insertion and
 * retreival, and full clearing in O(n) time, and freeing
 * a single vector in O(1) time.
 *
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#include "vecvec.h"
#include "vec.h"
#include <stdio.h>
#include <string.h>

static vec_cell vectors[MAX_VECS];
static int INITIALIZED = 0;

/**
 * @brief Marks each vector in the vector struct
 * as open.
 *
 * @return int
 */
int clear_vectors() {
    int cleared = 0;
    for(int i = 0; i < 10; i++) {
        if(vectors[i].state == CLOSED) {
            vectors[i].state = OPEN;
            cleared++;
        }
    }
    return cleared;
}

/**
 * @brief Prints a list of all stored vector
 *
 */
void list_vectors() {
    printf("Stored Vectors: \n");
    for(int i = 0; i < 10; i++) {
        if(vectors[i].state == CLOSED) {
            printf("\t%s: %s\n", vectors[i].name, vector_to_string(vectors[i].vec)
);
        }
    }
}

/**
 * @brief Inserts a vector into the vector array as a named variable
 *
 * @param vec Vector to store
 * @param name Variable name
 * @return int
 */
int insert_vector(vector vec, char* name) {
    if(!INITIALIZED) {
        clear_vectors();
        INITIALIZED = 1;
```

```
64      }
65
66      if (strlen(name) > 12) {
67          printf("Error: Variable name must be 12 characters or less\n");
68          return -1;
69      }
70
71      for (int i = 0; i < MAX_VECS; i++) {
72          if(vectors[i].state == OPEN) {
73              vectors[i].state = CLOSED;
74              vectors[i].vec = vec;
75              strcpy(vectors[i].name, name);
76              return 0;
77          } else if(!strcmp(vectors[i].name, name)) {
78              vectors[i].vec = vec;
79              strcpy(vectors[i].name, name);
80              return 0;
81          }
82      }
83      printf("Error: Vector storage is full. Please free a variable.\n");
84      return -1;
85  }
86
87  /**
88   * @brief Get a named vector pointer from the vector array
89   *
90   * @param name Name of the vector
91   * @return vec_cell*
92   */
93  vec_cell* get_vector(char* name) {
94      for(int i = 0; i < MAX_VECS; i++) {
95          if(!strcmp(name, vectors[i].name) && vectors[i].state == CLOSED) {
96              return &vectors[i];
97          }
98      }
99      return NULL;
100 }
101
102 /**
103  * @brief Marks a vector as free
104  *
105  * @param name Name of vector to free
106  * @return int
107  */
108 int free_vector(char* name) {
109     printf("Freeing %s\n", name);
110     for(int i = 0; i < MAX_VECS; i++) {
111         if(!strcmp(name, vectors[i].name)) {
112             vectors[i].state = OPEN;
113             return 0;
114         }
115     }
116     return -1;
117 }
118
119 /**
120  * @brief Converts a vec_cell to a string
121  *
122  * @param v
123  * @return char*
124  */
125 char* vec_cell_to_string(vec_cell* v) {
126     static char buffer[80];
127     snprintf(buffer, 80, "%s: %s", v->name, vector_to_string(v->vec));
```

```
128         return buffer;
129     }
```

```c
/**
 * @file ast.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief Helper routines for parsing an input string, constructing
 * an abstract syntax tree according to context free grammar G,
 * and evaluating the tree to a final result. Assignments are expressions,
 * not statements, and evaluate to the left hand side.
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#ifndef AST_H
#define AST_H
    #include "vec.h"

    typedef enum {
        TOKEN_IDENTIFIER,
        TOKEN_EQUALS,
        TOKEN_COMMA,
        TOKEN_PLUS,
        TOKEN_MINUS,
        TOKEN_STAR,
        TOKEN_SLASH,
        TOKEN_END,
        TOKEN_LPAREN,
        TOKEN_RPAREN,
        TOKEN_LBRACKET,
        TOKEN_RBRACKET,
        TOKEN_DOT,
        TOKEN_CROSS,
        TOKEN_CONST
    } token_type;

    typedef struct {
        token_type type;
        char* name;
    } token;

    typedef enum {
        NODE_ASSIGNMENT,
        NODE_OPERATION,
        NODE_IDENTIFIER,
        NODE_VECTOR,
        NODE_CONSTANT,
    } node_type;

    typedef struct node node;
    struct node {
        char* value;
        node_type type;
        int is_root;
        node* left;
        node* right;
    };


    typedef enum {
        VAL_VECTOR,
        VAL_SCALAR,
        VAL_SENTINEL,
    } value_type;
```

```
65
66      typedef struct {
67          value_type type;
68          union {
69              float scalar;
70              vector vec;
71          };
72      } value;
73
74      node* parse_input(char* input);
75      void print_ast(node* root);
76      void free_ast(node* root);
77      value evaluate_ast(node* n);
78      char* value_to_string(value v);
79      void print_help();
80
81  #endif
```

```c
/**
 * @file ast.c
 * @author Caleb Andreano (andreanoc@msoe.edu)
 * @class CPE2600-121
 * @brief Helper routines for parsing an input string, constructing
 * an abstract syntax tree according to context free grammar G,
 * and evaluating the tree to a final result. Assignments are expressions,
 * not statements, and evaluate to the left hand side.
 *
 *
 * Let G :=
 *  1. <statement> := <statement><statement>
 *  2. <statement> := <assignment> | <expression>
 *  3. <assignment> := <identifier> = <expression>
 *  4. <expression> := <term> | <term> { + | - } <expression>
 *  5. <term> := <factor> | <factor> { * | / | .| X } <term>
 *  6. <factor> := <identifier> | <vector> | <constant> |(<expression>)
 *  7. <identifier> := [a-zA-Z]+
 *  8. <value> := { <constant> | <constant>, <constant>, <constant> }
 *
 * Course: CPE2600-121
 * Assignment: Lab Wk 5
 * @date 2023-10-01
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <float.h>

#include "ast.h"
#include "vec.h"
#include "vecvec.h"
#include "tritone.h"

/**
 * @brief Returns the next valid token in the input buffer
 * at positon position
 *
 * @param input input string
 * @param position current position in the string
 * @return token
 */
token find_next_token(char* input, int* position) {

    char cur = input[(*position)];
    while(isspace(cur)) {
        cur = input[++(*position)];
    }

    token tok;
    tok.name = NULL;

    // Switch case on characters: Single character operators or identifiers
    switch(cur) {
        case '+':
            tok.name = "+";
            tok.type = TOKEN_PLUS;
            (*position)++;
            break;
        case '-':
            tok.name = "-";
            tok.type = TOKEN_MINUS;
```

```
65                  (*position)++;
66                  break;
67              case '*':
68                  tok.name = "*";
69                  tok.type = TOKEN_STAR;
70                  (*position)++;
71                  break;
72              case '/':
73                  tok.name = "/";
74                  tok.type = TOKEN_SLASH;
75                  (*position)++;
76                  break;
77              case '.':
78                  tok.name = ".";
79                  tok.type = TOKEN_DOT;
80                  (*position)++;
81                  break;
82              case 'X':
83                  tok.name = "X";
84                  tok.type = TOKEN_CROSS;
85                  (*position)++;
86                  break;
87              case '=':
88                  tok.name = "=";
89                  tok.type = TOKEN_EQUALS;
90                  (*position)++;
91                  break;
92              case ',':
93                  tok.name = ",";
94                  tok.type = TOKEN_COMMA;
95                  (*position)++;
96                  break;
97              case '\0':
98                  tok.name = "EOF";
99                  tok.type = TOKEN_END;
100                 (*position)++;
101                 break;
102             case '(':
103                 tok.name = "(";
104                 tok.type = TOKEN_LPAREN;
105                 (*position)++;
106                 break;
107             case ')':
108                 tok.name = ")";
109                 tok.type = TOKEN_RPAREN;
110                 (*position)++;
111                 break;
112             case '{':
113                 tok.name = "{";
114                 tok.type = TOKEN_LBRACKET;
115                 (*position)++;
116                 break;
117             case '}':
118                 tok.name = "}";
119                 tok.type = TOKEN_RBRACKET;
120                 (*position)++;
121                 break;
122             default:
123                 // Identifiers must start with a letter and then can be alphanumeric
124                 if(isalpha(cur)) {
125                     tok.type = TOKEN_IDENTIFIER;
126
127                     int size = 1;
128
```

```
129                     while(isalnum(input[*position + size])) {
130                         size++;
131                     }
132 
133                     // malloc the length of the identiifier + 1 for null terminator
134                     tok.name = malloc(size + 1);
135                     // copy name into malloced space
136                     memcpy(tok.name, (input + (*position)), size);
137                     tok.name[size] = '\0';
138                     // advance the position pointer
139                     (*position) += size;
140                 // Constants always are numbers
141                 } else if (isdigit(cur) || cur == '.') {
142                     // Parse numbers (integer or floating-point)
143                     tok.type = TOKEN_CONST;
144 
145                     int size = 1;
146 
147                     while (isdigit(input[*position + size])
148                         || input[*position + size] == '.') {
149                         size++;
150                     }
151 
152                     // malloc the string length of the number + \0
153                     tok.name = malloc(size + 1);
154                     memcpy(tok.name, (input + (*position)), size);
155                     tok.name[size] = '\0';
156                     // advance the position pointer
157                     (*position) += size;
158 
159                 } else {
160                     printf("Invalid token %c at position %d, ignoring\n",
161                         input[*position], *position);
162                 }
163     }
164     return tok;
165 }
166 
167 /**
168  * @brief Lexes the input string and returns a list of valid tokens
169  *
170  * @param input Input string
171  * @return token*
172  */
173 token* lex(char* input) {
174     static int capacity = 100;
175     int position = 0;
176     int size = 0;
177     // maximum number of tokens is hardcoded to 100;
178     token* tokens = malloc(capacity * sizeof(token));
179 
180     while(1) {
181         token tok = find_next_token(input, &position);
182 
183         tokens[size++] = tok;
184 
185         if(tok.type == TOKEN_END) {
186             break;
187         }
188     }
189     return tokens;
190 }
191 
192 
```

```c
/**
 * @brief Create a node struct
 *
 * @param type Type of node
 * @param value Node value
 * @param left Left Child
 * @param right Right Child
 * @return node*
 */
node* create_node(node_type type, char* value, node* left, node* right) {
    node* n = (node*) malloc(sizeof(node));
    n->type = type;
    n->value = value;
    n->left = left;
    n->right = right;
    n->is_root = 0;
    return n;
}


static node* parse_statement(token *tokens, int *position);
static node* parse_expression(token *tokens, int *position);
static node* parse_term(token *tokens, int *position);
static node* parse_factor(token *tokens, int *position);
static node* parse_identifier(token *tokens, int *position);
static node* parse_constant(token *tokens, int *position);
static node* parse_value(token *tokens, int *position);
static node* parse_assignment(token* tokens, int* position);

/**
 * @brief Lexes and parses an input string according to G
 *
 * @param input
 * @return node*
 */
node* parse_input(char* input) {
    token* tokens = lex(input);
    int position = 0;
    return parse_statement(tokens, &position);
}

/**
 * @brief Parses a statement and returns its root node
 *  <statement> := <assignment> | <expression>
 *
 * @param tokens
 * @param position
 * @return node*
 */
static node* parse_statement(token *tokens, int* position) {
    if(tokens[*position].type == TOKEN_IDENTIFIER
        && tokens[*position + 1].type == TOKEN_EQUALS) {
        return parse_assignment(tokens, position);
    } else if(tokens[*position].type == TOKEN_EQUALS) {
        printf("Error: assignment with no identifier\n");
        return NULL;
    } else {
        return parse_expression(tokens, position);
    }
}

/**
 * @brief
 * Parses an assignment and returns its root node
```

```
257  * <assignment> := <identifier> = <expression>
258  * @param tokens
259  * @param position
260  * @return node*
261  */
262 static node* parse_assignment(token* tokens, int* position) {
263     node* identifier = parse_identifier(tokens, position);
264     (*position)++;
265     return create_node(
266         NODE_ASSIGNMENT,
267         "=",
268         identifier,
269         parse_expression(tokens, position)
270     );
271 }
272
273 static node* parse_expression(token* tokens, int* position) {
274     node* term = parse_term(tokens, position);
275     while(tokens[*position].type == TOKEN_PLUS
276        || tokens[*position].type == TOKEN_MINUS) {
277         char* operator = tokens[(*position)].name;
278         (*position)++;
279         node* right = parse_term(tokens, position);
280         term = create_node(NODE_OPERATION, operator, term, right);
281     }
282     return term;
283 }
284
285 /**
286  * @brief Parses a term and returns it's root node
287  * <term> := <factor> | <factor> { * | / | .| X } <term>
288  * @param tokens
289  * @param position
290  * @return node*
291  */
292 static node* parse_term(token* tokens, int* position) {
293     node* factor = parse_factor(tokens, position);
294
295     while(
296             tokens[*position].type == TOKEN_STAR ||
297             tokens[*position].type == TOKEN_SLASH ||
298             tokens[*position].type == TOKEN_CROSS ||
299             tokens[*position].type == TOKEN_DOT
300         ) {
301         char* operator = tokens[(*position)].name;
302         (*position)++;
303         node* right = parse_factor(tokens, position);
304         factor = create_node(NODE_OPERATION, operator, factor, right);
305     }
306     return factor;
307 }
308
309 /**
310  * @brief
311  * Parses a factor and returns its root node
312  * <factor> -> <id> | V | (<exp>)
313  * @param tokens
314  * @param position
315  * @return node*
316  */
317 static node* parse_factor(token* tokens, int* position) {
318     if(tokens[*position].type == TOKEN_LPAREN) {
319         (*position)++;  // consume (
320         node* expression = parse_expression(tokens, position);
```

```
321             (*position)++;  // consume ()
322             return expression;
323         } else if(tokens[*position].type == TOKEN_IDENTIFIER) {
324             return parse_identifier(tokens, position);
325         } else if(tokens[*position].type == TOKEN_CONST) {
326             return parse_value(tokens, position);
327         } else {
328             printf("Error at position %d near token '%s'\n",
329                 *position, tokens[*position-1].name);
330             return NULL;
331         }
332     }
333
334     /**
335      * @brief
336      * Parses a constant value
337      * @param tokens
338      * @param position
339      * @return node*
340      */
341     static node* parse_constant(token* tokens, int* position) {
342         char* value = tokens[(*position)].name;
343         (*position )++;
344         return create_node(NODE_CONSTANT, value, NULL, NULL);
345     }
346
347
348     /**
349      * @brief
350      *  Parses a value and returns its root node
351      *  V -> { <const> | <const>, <const>, <const> }
352      * @param tokens
353      * @param position
354      * @return node*
355      */
356     static node* parse_value(token* tokens, int* position) {
357         if(tokens[*position].type == TOKEN_CONST) {
358             node* i = parse_constant(tokens, position);
359             if(tokens[*position].type == TOKEN_COMMA) {
360                 (*position)++;
361             }
362
363             node* j;
364             node* k;
365
366             token next = tokens[*position];
367             // If there's two constants in a row
368             if(next.type != TOKEN_END && next.type == TOKEN_CONST) {
369                 j = parse_constant(tokens, position);
370
371                 if(tokens[*position].type == TOKEN_COMMA) {
372                     (*position)++;
373                 }
374
375                 // If there's three constants
376                 if(tokens[*position].type != TOKEN_END) {
377                     k = parse_constant(tokens, position);
378                     if(tokens[*position].type == TOKEN_COMMA) {
379                         (*position)++;
380                     }
381
382                     // If there's four right night to eachother
383                     if(tokens[*position].type == TOKEN_CONST) {
384                         printf("Warning: Only 3D vectors are supported. Using"
```

```
385                                     " first three tokens.\n");
386                 };
387
388             } else {
389                 k = create_node(NODE_CONSTANT, "0", NULL, NULL);
390             }
391             return create_node(NODE_VECTOR, NULL, i, create_node(NODE_VECTOR, NU
    LL, j, k));
392         } else {
393             // there's only one constant
394             return i;
395         }
396     } else {
397         return NULL;
398     }
399 }
400
401
402
403
404 /**
405  * @brief Parses an identifier and returns it's node
406  *
407  * <identifier> := [a-zA-Z]+
408  * @param tokens
409  * @param position
410  * @return node*
411  */
412 static node* parse_identifier(token* tokens, int* position) {
413     char* name = tokens[(*position)].name;
414     (*position)++;
415     return create_node(NODE_IDENTIFIER, name, NULL, NULL);
416 }
417
418
419 /**
420  * @brief Recursively frees a tree given its root node
421  *
422  * @param n root node
423  */
424 void free_ast(node* n) {
425     if (n == NULL) {
426         return;
427     }
428
429     free_ast(n->left);
430     free_ast(n->right);
431     free(n);
432 }
433
434 /**
435  * @brief Recursively prints an AST node given its depth
436  *
437  * @param node
438  * @param depth
439  */
440 static void print_ast_recursive(node* node, int depth) {
441     if (node == NULL) {
442         return;
443     }
444
445     // Print indentation based on depth
446     for (int i = 0; i < depth; ++i) {
447         printf(" ");
```

```
448         }
449
450         // Print node information
451         printf("Type: %d, Value: %s\n", node->type, node->value);
452
453         // Recursively print left and right children
454         print_ast_recursive(node->left, depth + 1);
455         print_ast_recursive(node->right, depth + 1);
456     }
457
458     /**
459      * @brief Recursively prints a tree given it's root node
460      *
461      * @param root
462      */
463     void print_ast(node* root) {
464         printf("Abstract Syntax Tree:\n");
465         print_ast_recursive(root, 0);
466     }
467
468     /**
469      * @brief Converts a vector to a value struct
470      *
471      * @param v
472      * @return value
473      */
474     static value make_value_from_vector(vector v) {
475         value r;
476         r.type = VAL_VECTOR;
477         r.vec = v;
478         return r;
479     }
480
481     /**
482      * @brief Converts a scalar type to a struct
483      *
484      * @param f
485      * @return value
486      */
487     static value make_value_from_scalar(float f) {
488         value r;
489         r.type = VAL_SCALAR;
490         r.scalar = f;
491         return r;
492     }
493
494     /**
495      * @brief Returns the sentinel value struct
496      *
497      * @return value
498      */
499     static value sentinel() {
500         value r ;
501         r.type = VAL_SENTINEL;
502         return r;
503     }
504
505     /**
506      * @brief returns true if a value is the sentinel value
507      *
508      * @param s
509      * @return int
510      */
511     static int is_sentinel(value s) {
```

```
512     return s.type == VAL_SENTINEL;
513 }
514
515 /**
516  * @brief Converts a value to a string and returns it
517  *
518  * @param v
519  * @return char*
520  */
521 char* value_to_string(value v) {
522     static char buffer[200];
523     if(!is_sentinel(v)) {
524         if(v.type == VAL_VECTOR) {
525             snprintf(buffer, 200, "%s\n", vector_to_string(v.vec));
526         } else {
527             snprintf(buffer, 200, "%.2f\n", v.scalar);
528         }
529     } else {
530         snprintf(buffer, 200, "%s", "");
531     }
532     return buffer;
533
534 }
535
536 /**
537  * @brief Evaluates an assignment node and returns it's value
538  *
539  * @param n
540  * @return value
541  */
542 static value handle_asssignemnt(node* n) {
543     if(n->left == NULL || n->left->type != NODE_IDENTIFIER || n->right == NULL)
    {
544         return sentinel();
545     }
546
547     value result = evaluate_ast(n->right);
548     if(!is_sentinel(result)) {
549         if(result.type == VAL_VECTOR) {
550             insert_vector(result.vec, n->left->value);
551             return result;
552         } else {
553             printf("Warning: Cannot assign scalar to variable\n");
554             printf("Assigning scalar as field i\n");
555             vector v = {result.scalar, 0, 0};
556             if(insert_vector(v, n->left->value) != -1) {
557                 value r;
558                 r.type = VAL_VECTOR;
559                 r.vec = v;
560                 return r;
561             } else {
562                 return sentinel();
563             };
564         }
565     } else {
566         return sentinel();
567     }
568 }
569
570 /**
571  * @brief Handles identifiers nodes and processes relevant commands.
572  * If the identifier is not a command, returns the value, otherwise sentinel
573  *
574  * @param n
```

```
575  * @return value
576  */
577 static value handle_identifier(node* n) {
578     if(!strcmp(n->value, "quit")) {
579         exit(0);
580     } else if(!strcmp(n->value, "free")) {
581         int cleared = clear_vectors();
582         printf("Freed %d vectors\n", cleared);
583         return sentinel();
584     } else if(!strcmp(n->value, "list")) {
585         list_vectors();
586         return sentinel();
587     } else if(!strcmp(n->value, "help")) {
588         print_help();
589         return sentinel();
590     } else if(!strcmp(n->value, "clear")) {
591         printf("\033[2J"); // clear screen
592         printf("\033[H"); // go home
593 
594         return sentinel();
595     } else {
596         vec_cell* v = get_vector(n->value);
597         if(v != NULL) {
598             return make_value_from_vector(v->vec);
599         } else {
600             printf("Error: no vector found named %s\n", n->value);
601             return sentinel();
602         }
603     }
604 }
605 
606 /**
607  * @brief Handles vector and scalar operation nodes and returns their value
608  *
609  * @param n
610  * @return value
611  */
612 static value handle_operation(node*n) {
613     value left = evaluate_ast(n->left);
614     value right = evaluate_ast(n->right);
615 
616     // Addition operations
617     if(!strcmp(n->value, "+")) {
618 
619         if(left.type == VAL_VECTOR && right.type == VAL_VECTOR) {
620             vector sum = vec_add(left.vec, right.vec);
621             return make_value_from_vector(sum);
622         } else if(left.type == VAL_SCALAR && right.type == VAL_SCALAR) {
623             float sum = left.scalar + right.scalar;
624             value v = make_value_from_scalar(sum);
625             return v;
626         } else {
627             printf("Error: addition not implemented for scalar + vector\n");
628             return sentinel();
629         }
630     // Subtraction Operations
631     } else if(!strcmp(n->value, "-")) {
632 
633         if(left.type == VAL_VECTOR && right.type == VAL_VECTOR) {
634             vector sum = vec_sub(left.vec, right.vec);
635             return make_value_from_vector(sum);
636         } else if(left.type == VAL_SCALAR && right.type == VAL_SCALAR) {
637             float sum = left.scalar - right.scalar;
638             return make_value_from_scalar(sum);
```

```
639             } else {
640                 printf("Error: subtraction not implemented for scalar + vector\n");
641                 return sentinel();
642             }
643         // Multiplicaton functions
644         } else if(!strcmp(n->value, "*")) {
645 
646             if(left.type == VAL_VECTOR && right.type == VAL_VECTOR) {
647                 vector sum = vec_mul(left.vec, right.vec);
648                 return make_value_from_vector(sum);
649             } else if(left.type == VAL_SCALAR && right.type == VAL_SCALAR) {
650                 float sum = left.scalar * right.scalar;
651                 return make_value_from_scalar(sum);
652             } else if(left.type == VAL_VECTOR && right.type == VAL_SCALAR) {
653                 float i = left.vec.i * right.scalar;
654                 float j = left.vec.j * right.scalar;
655                 float k = left.vec.k * right.scalar;
656                 vector product = {i, j, k};
657                 return make_value_from_vector(product);
658             } else {
659                 float i = right.vec.i * left.scalar;
660                 float j = right.vec.j * left.scalar;
661                 float k = right.vec.k * left.scalar;
662                 vector product = {i, j, k};
663                 return make_value_from_vector(product);
664             }
665         // Division operations
666         } else if(!strcmp(n->value, "/")) {
667 
668             if(left.type == VAL_SCALAR && right.type == VAL_SCALAR) {
669                 return make_value_from_scalar(left.scalar/right.scalar);
670             } else {
671                 printf("Error: invalid arguments to scalar division\n");
672                 return sentinel();
673             }
674         // Dot produt
675         } else if(!strcmp(n->value, ".")) {
676 
677             if(left.type == VAL_VECTOR && right.type == VAL_VECTOR) {
678                 float sum = vec_dot(left.vec, right.vec);
679                 return make_value_from_scalar(sum);
680             } else {
681                 printf("Error: invalid arguments to dot product\n");
682                 return sentinel();
683             }
684         // Cross product
685         } else if(!strcmp(n->value, "X")) {
686 
687             if(left.type == VAL_VECTOR && right.type == VAL_VECTOR) {
688                 vector cross = vec_cross(left.vec, right.vec);
689                 return make_value_from_vector(cross);
690             } else {
691                 printf("Error: invalid arguments to cross product\n");
692                 return sentinel();
693             }
694         } else {
695             return sentinel();
696         }
697     }
698 
699     /**
700      * @brief Handles constructing vectors from vector root nodes:
701      * Vectors in memory are represented as:
702      *
```

```
 *               NODE_VECTOR: null
 *                  /          \
 *                 /            \
 *      NODE_CONSTANT: i         NODE_VECTOR: null
 *                                 /            \
 *                    NODE_CONSTANT: j     NODE_CONSTANT: k
 *
 * @param n
 * @return value
 */
static value handle_vector(node* n) {
    float i = atof(n->left->value);
    float j = atof(n->right->left->value);
    float k = atof(n->right->right->value);
    vector v = {i, j, k};
    return make_value_from_vector(v);
}

/**
 * @brief Evaluates an AST branch given the root node using
 * recursive descent parsing (essentially pre-order traversal).
 * Returns its value.
 *
 * @param n
 * @return value
 */
value evaluate_ast(node* n) {
    if(n == NULL) {
        return sentinel();
    }

    switch(n->type) {
        case(NODE_OPERATION):
            return handle_operation(n);
            break;
        case(NODE_IDENTIFIER):
            return handle_identifier(n);
            break;
        case(NODE_ASSIGNMENT):
            return handle_asssignemnt(n);
            break;
        case(NODE_VECTOR):
            return handle_vector(n);
            break;
        case(NODE_CONSTANT):
            return make_value_from_scalar(atof(n->value));
        default:
            return sentinel();
    }
}
```