Sinan DÖŞEYİCİ
05180000037

Department of Engineering, Computer Engineering
EGE University

Project-2 PL Interpreter/ 2021

GENERAL

Developing an interpreter on top of a lexical analyzer for Big Add Language.

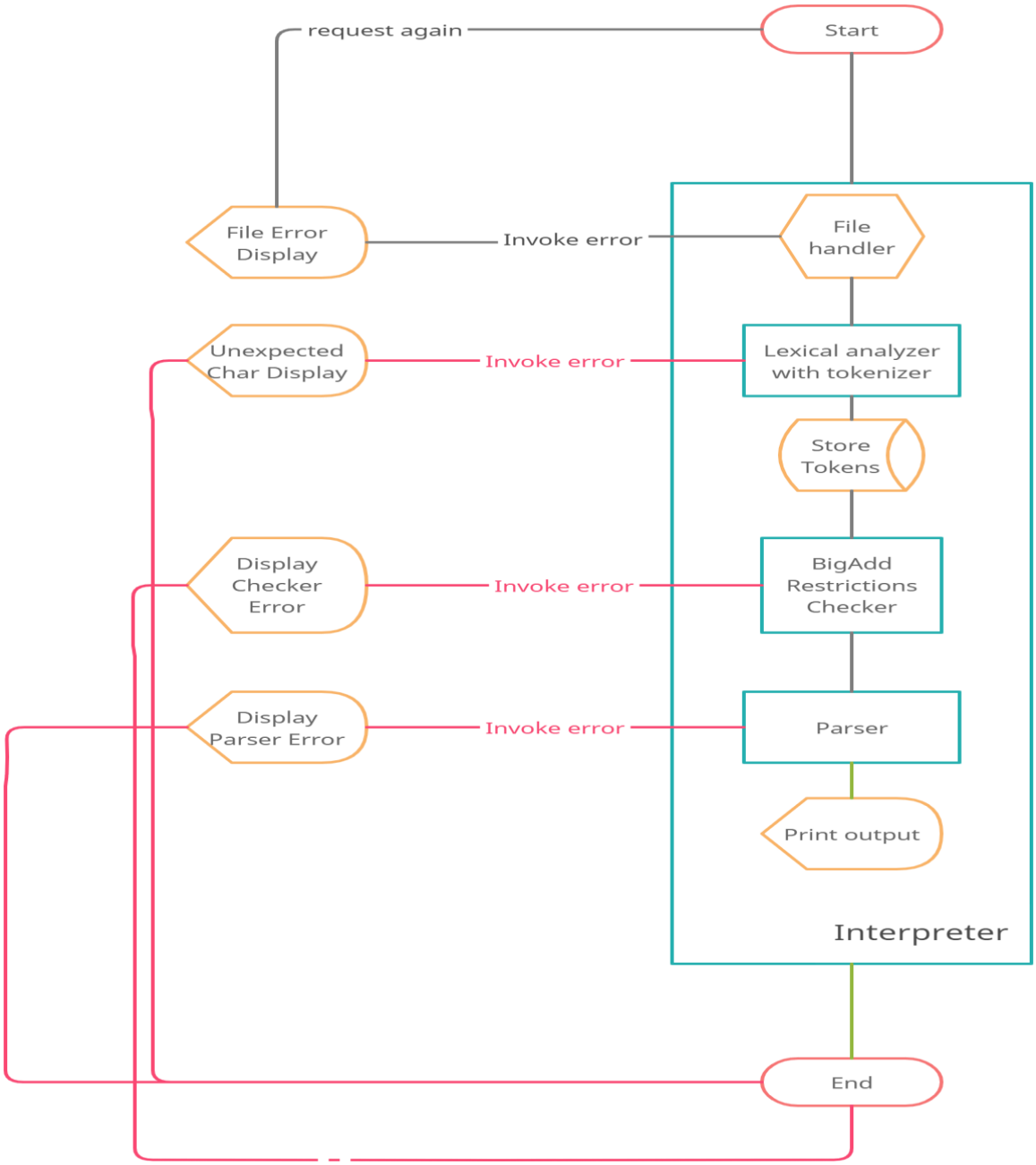
CONTENTS

PLATFORM

Written for C compiled with MinGW-Windows-GCC,

1. Analysis and Expectations

The project involves developing an interpreter for Big Add. Big Add is capable of a variable declaration, an assignment statement, loop assignment, iterations, adding, subtraction, and printing strings as an output.



2. Design and Source Code

2.a) File Handler

The file handler code opens files and manages input-output stream arguments.

```
  // IO Stream part

//.....
------------------------------------------------------------------
  if (argc <= 1) {
    filename = malloc(100 * sizeof(char));
    *(filename) ='\0';
  } else if (argc == 2)
    filename =argv[1];
  else {
    //program working path always passed as first argument
    printf("[FileHandler] Too many arguments given. Maximum one argument
expected.");
    return stop();
  }
  while (true) { //loop until user enters a correct filename
    if (argc <= 1 || cont) {
      printf("Enter a file name: ");
      scanf(" %[^\n]s", filename); //to accept spaces in file name too
    } else {
      cont = true;
    }
    bool filename_ok = false;
    if (strstr(filename, ".") != NULL) {
      strrev(filename);
      if (filename[0] == 'a' && filename[1] == 'b' && filename[2] == '.') {
        strrev(filename);
        filename_ok = true;
      } else {
        printf("[FileHandler] Error: Wrong extension!\n");
      }
    } else { //filename dont have extension, lets add
      int len = (int) strlen(filename);
```

```
      filename[len] = '.';
      filename[len + 1] = 'b';
      filename[len + 2] = 'a';
      filename[len + 3] = '\0';
      filename_ok = 1;
    }
    if (filename_ok) {
      if (access(filename, F_OK) == -1) {
        printf("[FileHandler] File doesn't exist\n");
      } else {
        break; // file exist, break the while loop
      }
    }
  }
  // end of IOStream part

//------------------------------------------------------------------------
```

2.b) Lexical analyzer with tokenizer (Can return Critical Error)

Opens a file and reads characters at a time. The incoming value should be tokenized and if the analyzer encounters the not recognizable character, it returns -1 as a critical error and exits.

```
  //lexical analyzer with tokens
  // @returns token array

//------------------------------------------------------------------

  FILE * source_code = fopen(filename, "r");
  //temp char array for lexical analyzer.
  char lexeme[105];
  struct token tokens[1000];
  int token_count =0;
  int eof_col = 0;

  int i = 0;
  int col = 0;
  int start_column = 0;
  bool is_reading_comment = false;
```

```
bool is_reading_string = false;
bool is_integer =true;
while (1) {
  char c = (char) fgetc(source_code);

  if (c == '{' && !is_reading_comment && !is_reading_string) {
    //fputs("{ is a parenthesis\n", output_file);
    is_reading_comment = true;
    continue;
  } else if (c == '}' && is_reading_comment && !is_reading_string) {
    //fputs("} is a parenthesis\n", output_file);
    is_reading_comment = false;
    continue;
  }

  col++;
  if (i == 0) {
    start_column =col;
  }
  if (c == '\n') {
    eof_col = col;
    col = 0;
    line_count++;
  }

  if (!is_reading_comment) {
    if (c == '"') {
      if (is_reading_string) {
        lexeme[i] ='\0';
        tokens[token_count].type = "string";
         //duplicate lexeme, we will change lexeme later
        tokens[token_count].value = strdup(lexeme);
        token_count++;
        i = 0;
      }
      is_reading_string = !is_reading_string;
      continue;
    }
    if (is_reading_string) {
      lexeme[i++] = c;
    }

    if (!is_reading_string) {
      if ((c >= 65 && c <= 91) || (c >= 97 && c <= 123) || (c >= 48 && c
```

```
<= 57) || (c >= 44 && c <= 46) ||
          c == 93 || c == 125 || c == 32 || c == 9 || c == 10 || c == 95 ||
c == -1) {
          //accept A-Z a-z 0-9 , - . [] {} space \t \n _ characters in
source code

          if (c == '[') {
            tokens[token_count].type = "parenthesis";
            tokens[token_count].value = "[";
            tokens[token_count].line = line_count;
            tokens[token_count].column = start_column;
            token_count++;
            continue;
          } else if (c == ']') {
            tokens[token_count].type = "parenthesis";
            tokens[token_count].value = "]";
            tokens[token_count].line = line_count;
            tokens[token_count].column = start_column;
            token_count++;
            continue;
          }

          if (c != ' ' && c != '.' && c != '\t' && c != '\n' && c != ',' &&
c != EOF) {
            if ((c < 48 && c != 45) || c > 57) // 45='-'
              is_integer = false;

            lexeme[i++] = c; //still reading a lexeme
          } else {
            //check if lexeme is not null/empty
            if (i != 0) {
              lexeme[i] = '\0';
              bool is_keyword = false;
              //checker loop for keyword
              for (int j = 0; j < 10; j++) {
                if (strcmp(lexeme, keywords[j]) == 0) {
                  tokens[token_count].type = "keyword";
                  tokens[token_count].value = strdup(lexeme);
                  tokens[token_count].line = line_count;
                  tokens[token_count].column = start_column;
                  token_count++;
                  is_keyword = true;
                  break;
                }
```

```
            }

            if (!is_keyword) {
              if (is_integer) {
                tokens[token_count].type = "integer";
                tokens[token_count].value = strdup(lexeme);
                tokens[token_count].line = line_count;
                tokens[token_count].column = start_column;
                token_count++;
              } else {
                tokens[token_count].type = "identifier";
                tokens[token_count].value = strdup(lexeme);
                tokens[token_count].line = line_count;
                tokens[token_count].column = start_column;
                token_count++;
              }
            }
          }
          if (c == '.') {
            tokens[token_count].type = "eol";
            tokens[token_count].value = ".";
            tokens[token_count].line = line_count;
            tokens[token_count].column = start_column;
            token_count++;
          } else if (c == ',') {
            tokens[token_count].type = "comma";
            tokens[token_count].value = ",";
            tokens[token_count].line = line_count;
            tokens[token_count].column = start_column;
            token_count++;
          }
          i = 0;
          is_integer = true; //reset

        }
      } else {
        printf("[Lexical Analyzer] Unexpected character: %c in line %d,
column %d", c, line_count, col);
        return stop();
      }
    }
  }
  if (c == EOF)
    break;
```

```
  }
  fclose(source_code);

  tokens[token_count].type = "end of file";
  tokens[token_count].value = "EOF";
  tokens[token_count].line = line_count;
  tokens[token_count].column = eof_col;
//-------------------------------------------------------------------------
```

2.c) BigAdd Restrictions Handler (Can return Critical Error)

Handles the BigAdd language restrictions.(Chapter 3) If it sees a limit exceeded and something prohibited, it closes the program with invoking general error invoker function.

```
struct stack p_stack;
p_stack.max = INITIAL_MAX;
p_stack.elements = NULL;
p_stack.top = -1;
int open_count = 0;
int close_count = 0;
int last_open = 0;
for (int l = 0; l < token_count; l++) {
  if (strcmp(tokens[l].type, "identifier") == 0) {
    if (strlen(tokens[l].value) >20)
      //max identifier name
      return error(7, NULL, tokens[l]);
    //if identifier contains -
    //not valid identifier
    if (strstr(tokens[l].value, "-") != NULL)
      return error(5,NULL, tokens[l]);
    //if first char is digit not valid identf.
    if (tokens[l].value[0] >= 48 && tokens[l].value[0] <= 57)
      return error(9, NULL, tokens[l]);
  } else if (strcmp(tokens[l].type, "integer") == 0) {
    int digit_limit;
    if (strstr(tokens[l].value, "-") != NULL)
      digit_limit = MAX_DIGIT + 1;
    else
      digit_limit = MAX_DIGIT;
    //Max digit limit exceeded
    if (strlen(tokens[l].value) > digit_limit)
```

```
      return error(4, NULL, tokens[l]);
    int dash_count = 0;
    char * temp = tokens[l].value;
    while (strstr(temp, "-") != NULL) {
      dash_count++;
      temp++;
    }
    // critic error of (--)123
    if (dash_count > 1)
      return error(6, NULL, tokens[l]);
  } else if (strcmp(tokens[l].type, "parenthesis") == 0) {
    if (strcmp(tokens[l].value, "[") == 0) {
      push(&p_stack, '[');
      open_count++;
      last_open = l;
    } else if (strcmp(tokens[l].value, "]") == 0) {
      char temp = pop( & p_stack);
      if (temp != '[')
        return error(1, "[CheckerBA]: Expected open parenthesis before
using a close parenthesis ", tokens[l]);
      close_count++;
    }
  }
}
if (open_count != close_count) {
  printf("[CheckerBA]: Expected a close parenthesis before end of file.
Last open parenthesis is on line %d",
    tokens[last_open].line);
  return stop();
}
//-----------------------------------------------------------------------------
```

2.d) Parser (Can return Critical Error)

Parser checks by iteration on the array named "tokens" returned by the lexical analyzer. If it encounters the identifier token, it assigns it to the symbol table struct we set up and if it sees a missing value error it will invoke the general error function and exit critically.

```
 //Parser
//-----------------------------------------------------------------------------
-------------------------------------------------------------
  struct token l_vars[100];
  int l_starts[100] = {
```

```
    0
  };
  int l_level = -1; //loop level, -1 means we are not in loop
  bool l_block[100] = {
    false
  }; //'true' if loop has code block, 'false' if it has one line code
  i = 0;

  //loop in tokens array. whole loop can be counted as parser
  //it interprets one line of code in every iteration!
  // used tokens[i + 1] or tokens[i + 2] for checking syntax
  // for ex: "move 5 to x." when we are on '5' token, checked next token if
its 'to'.
  // then increase i by 2. because we wont do anything with 'to' token.
  while (i < token_count) {
    // so everytime we reach here we have to check what type of line of
code are we going to read
    if (strcmp(tokens[i].type, "keyword") == 0 || strcmp(tokens[i].value,
"]") == 0) {
      if (strcmp(tokens[i].value, "int") == 0) { //new integer declaration
-> int x.
        i++;
        if (strcmp(tokens[i].type, "identifier") != 0)
          return error(1, "\n[Parser]: Expected an identifier.",
tokens[i]);

        if (strcmp(tokens[i + 1].type, "eol") != 0)
          return error(1, "\n[Parser]: Expected an end of line character",
tokens[i + 1]);
        //get() will return "not declared" if its not declared
        // so if its not returns "not declared" there is an error
        if (strcmp(get(tokens[i].value), "not declared") != 0)
          return error(3, NULL, tokens[i]);
        symbol_table[symbol_count].name = tokens[i].value;
        symbol_table[symbol_count].value = "0";
        symbol_count++;

        i += 2; //nothing to do with eol
      //checker of the keyword "move"
      } else if (strcmp(tokens[i].value, "move") == 0) {
        i++;
        if (strcmp(tokens[i].type, "identifier") != 0 &&
strcmp(tokens[i].type, "integer") != 0)
          return error(1, "\n[Parser]: Expected identifier or integer",
```

```
tokens[i]);

        if (strcmp(tokens[i + 1].value, "to") != 0)
          return error(1, "\n[Parser]: Expected keyword 'to'", tokens[i +
1]);

        if (strcmp(tokens[i + 2].type, "identifier") != 0)
          return error(1, "\n[Parser]: Expected an identifier", tokens[i]);
        //we can assign values to only identifiers

        if (strcmp(tokens[i + 3].type, "eol") != 0)
          return error(1, "\n[Parser]: Expected an end of line character",
tokens[i + 3]);

        //assignment syntax is correct
        char * new_val = valueof(tokens[i]);
        if (strcmp(new_val, "not declared") == 0)
          return error(2, NULL, tokens[i]);

        int found = set(tokens[i + 2].value, new_val); //returns 0 if
symbol not found
        if (!found)
          return error(2, NULL, tokens[i + 2]);

        i += 4;
     // checker of the addition
     } else if (strcmp(tokens[i].value, "add") == 0) {
        i++;
        if (strcmp(tokens[i].type, "identifier") != 0 &&
strcmp(tokens[i].type, "integer") != 0)
          return error(1, "\n[Parser]: Expected identifier or integer",
tokens[i]);

        if (strcmp(tokens[i + 1].value, "to") != 0)
          return error(1, "\n[Parser]: Expected keyword 'to'", tokens[i +
1]);

        if (strcmp(tokens[i + 2].type, "identifier") != 0)
          return error(1, "\n[Parser]: Expected an identifier", tokens[i +
2]);
        // we have to assign to a variable

        if (strcmp(tokens[i + 3].type, "eol") != 0)
          return error(1, "\n[Parser]: Expected an end of line character",
```

```
tokens[i + 3]);
        //addition syntax is correct

        char * new_val =valueof(tokens[i]);
        if (strcmp(new_val, "not declared") == 0)
          return error(2, NULL, tokens[i]);

        //target
        char * old_val = get(tokens[i + 2].value);
        if (strcmp(old_val, "not declared") == 0)
          return error(2, NULL, tokens[i + 2]);

        char * answer = add(old_val, new_val);
        if (strcmp(answer, "digit limit exceeded") == 0)
          return error(4, NULL, tokens[i + 2]);

        set(tokens[i + 2].value, answer);

        i += 4;
      // checker of the sub
      } else if (strcmp(tokens[i].value, "sub") == 0) {
        i++;
        if (strcmp(tokens[i].type, "identifier") != 0 &&
strcmp(tokens[i].type, "integer") !=0)
          return error(1, "\n[Parser]: Expected identifier or integer",
tokens[i]);

        if (strcmp(tokens[i + 1].value, "from") != 0)
          return error(1, "\n[Parser]: Expected keyword 'from'", tokens[i +
1]);

        if (strcmp(tokens[i + 2].type, "identifier") != 0)
          return error(1, "\n[Parser]: Expected an identifier", tokens[i +
2]);
        // we have to assign to a variable

        if (strcmp(tokens[i + 3].type, "eol") != 0)
          return error(1, "\n[Parser]: Expected an end of line character",
tokens[i + 3]);

        char * new_val =valueof(tokens[i]);
        if (strcmp(new_val, "not declared") == 0)
          return error(2, NULL, tokens[i]);
```

```
        //target
        char * old_val = get(tokens[i + 2].value);
        if (strcmp(old_val, "not declared") == 0)
          return error(2, NULL, tokens[i + 2]);

        char * answer = sub(old_val, new_val);
        if (strcmp(answer, "digit limit exceeded") == 0)
          return error(4, NULL, tokens[i + 2]);

        set(tokens[i + 2].value, answer);

        i += 4;
      //printer
      } else if (strcmp(tokens[i].value, "out") == 0) {
        i++;
        while (i < token_count) { //print everything till end of line
          if (strcmp(tokens[i].type, "string") == 0) {
            printf(tokens[i].value);
          } else if (strcmp(tokens[i].type, "identifier") == 0) {
            char * value = valueof(tokens[i]);
            if (strcmp(value, "not declared") == 0)
              return error(2, NULL, tokens[i]);

            printf(value);
          } else if (strcmp(tokens[i].value, "newline") == 0) {
            printf("\n");
          } else //its not printable
            return error(1, "\n[Parser]: Expected string, identifier or
'newline' keyword", tokens[i]);

          i++;

          if (strcmp(tokens[i].type, "eol") == 0)
            break;

          //if we reached here, we will continue printing. check if theres
a comma
          if (strcmp(tokens[i].type, "comma") != 0)
            return error(1, "\n[Parser]: Expected comma or end of line
character", tokens[i]);

          i++; //skipped comma
        }
     i++; //we were on end of line, check while loop condition
```

```
      } else if (strcmp(tokens[i].value, "loop") == 0) {
        i++;
        if (strcmp(tokens[i].type, "identifier") != 0 &&
strcmp(tokens[i].type, "integer") != 0)
          return error(1, "\n[Parser]: Expected identifier or integer",
tokens[i]);

        if (strcmp(tokens[i + 1].value, "times") != 0)
          return error(1, "\n[Parser]: Expected keyword 'times'", tokens[i
+ 1]);

        char * loop_count = valueof(tokens[i]);
        if (compare(loop_count, "1") == -1)
          return error(8, NULL, tokens[i]);

        l_level++;
        l_vars[l_level] = tokens[i];

        i += 2;
        if (strcmp(tokens[i].value, "[") == 0) {
          i++; // nothing to do with '['
          l_block[l_level] = true;
        } else if (strcmp(tokens[i].type, "keyword") != 0)
          return error(1, "\n[Parser]: Expected open paranthesis or a
keyword", tokens[i]);

        l_starts[l_level] = i;
        continue;
      }

      //interpreted a line of code, lets check if it was in loop
      if (l_level >= 0) {
        // contains loop
        if (l_block[l_level]) {
          if (strcmp(tokens[i].value, "]") == 0) {
            i++; //if its last iteration of loop, it will continue from
next line

          } else {
            // we are going to interpret atlease one line in current loop
            continue;
          }
        }
```

```
        char * old_val = valueof(l_vars[l_level]);
        char * new_val = sub(old_val, "1");

        if (strcmp(l_vars[l_level].type, "identifier") == 0) {
          set(l_vars[l_level].value, new_val);
        } else {
          l_vars[l_level].value = new_val;
        }
        if (strcmp(new_val, "0") == 0){
          l_starts[l_level] = 0;
          l_block[l_level] = false;
          l_level--;
        } else {
          i = l_starts[l_level];
        }
      }
    } else {

      return error(1, "\n[Parser]: Expected keyword", tokens[i]);
    }
  }

  printf("\n\nInterpreted succesfully! Press a key to exit...");
  fseek(stdin, 0, SEEK_END); //clear input buffer
  getchar();
  return 0;
}
```

3. Limitations

3.a) Lexical Analyzer Limitations

→ It does not accept any characters other than these characters. A-Z a-z 0-9 , - . [] {} space \t \n _

→ If it sees an unrecognized character, it exits.

3.b) BigAdd Restrictions Handler Limitations

→ Identifier can't start with a digit.

→ The maximum identifier length is 20.

→ Identifier can't contain char '-'.

→ The maximum integer limit is 100.

→ Integer cannot contain more than 1 of char '-'.

→ Open parentheses handler (Handles with struct Stack)

→ If it sees an over-limit or restricted identifier name, it exits.

3.c) Parser Limitations

→ It must have end of line character '.'

When it accepts a keyword, it checks the identifier and declaration within the declaration. (Restricted double declaration or no identifier name and checks the keyword 'to')

→ If accepts the keyword 'add' or 'sub', it can expect integers or identifiers with the keyword 'from' or 'to', otherwise it is restricted.

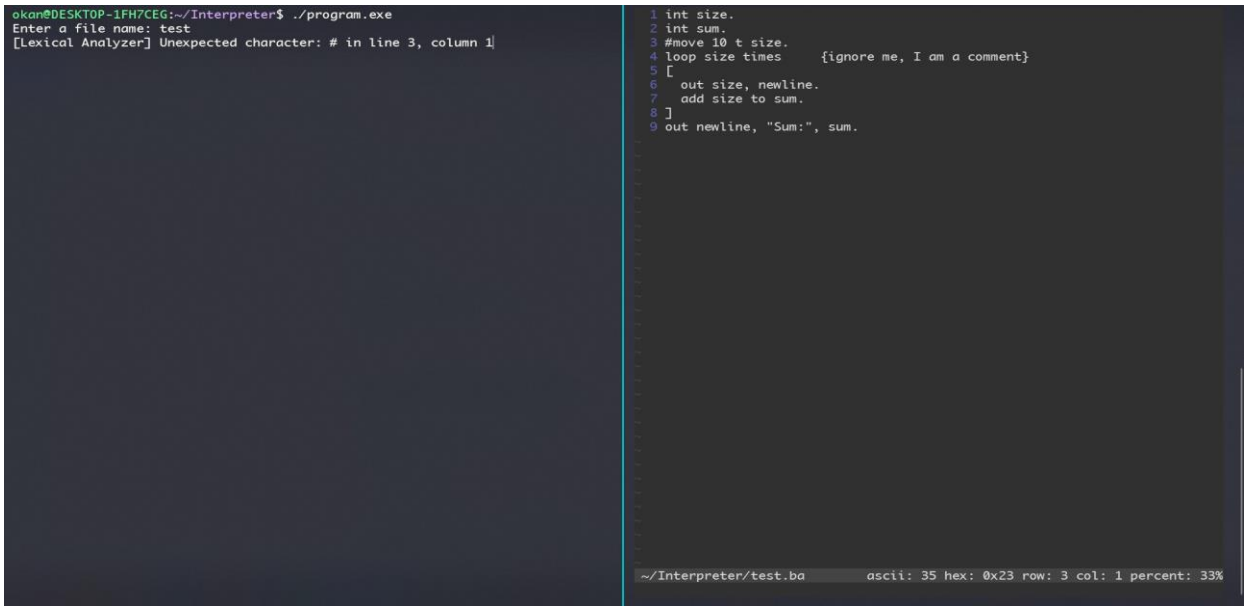
→ If accepts keyword 'out' it should wait for the next (Expected string, identifier or 'newline' keyword" otherwise it is restricted)

The keyword loop expects an identifier, an integer, and the keyword 'times', or otherwise, it is restricted.

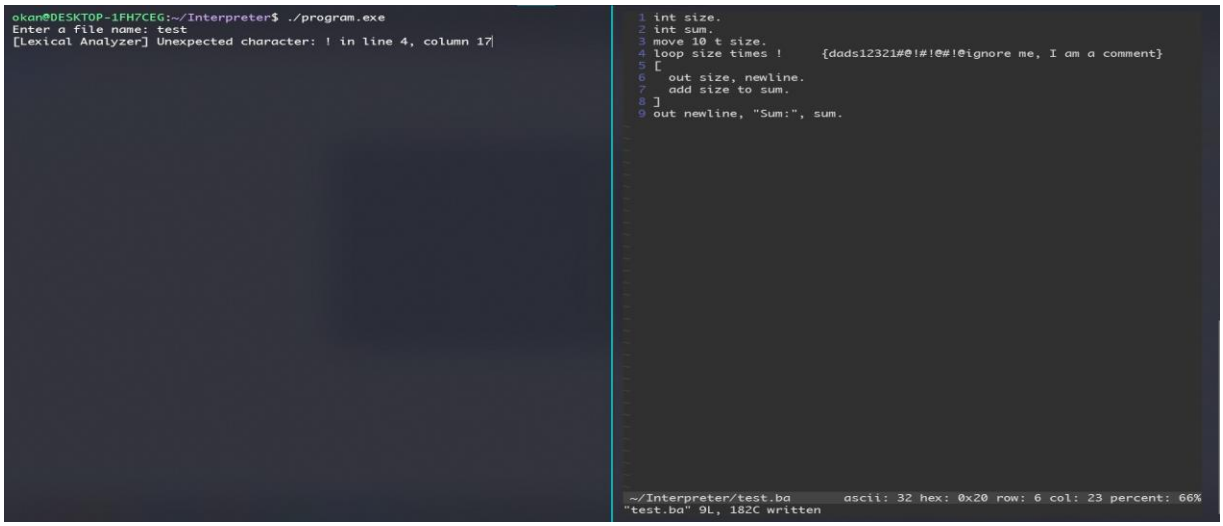
4. Testing

4.a Lexical Analyzer Testing

Unexpected Char Test

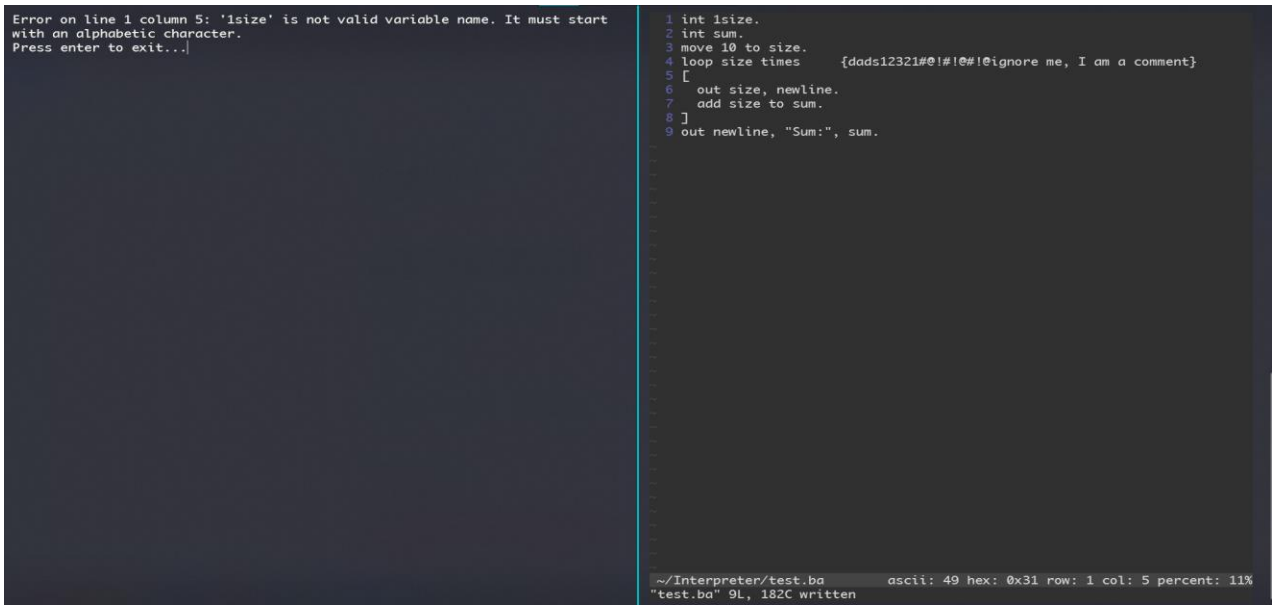


Identifier Limit Test

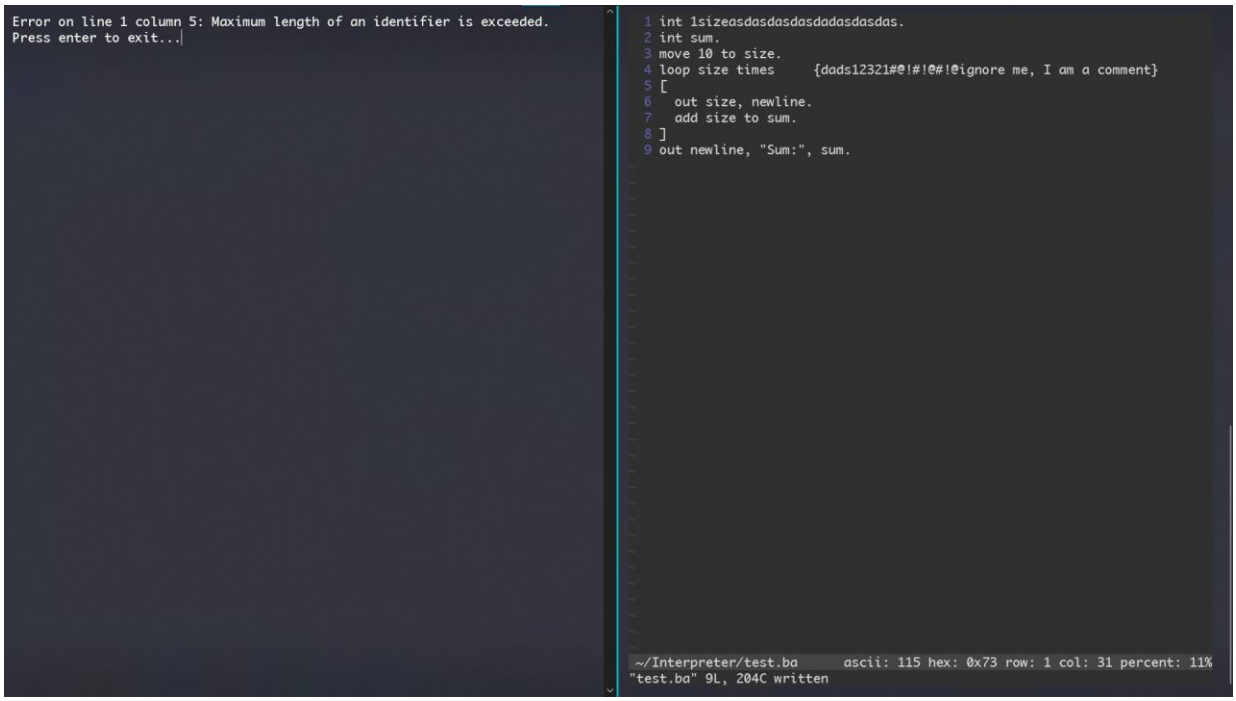


4.b BigAdd Restrictions Test

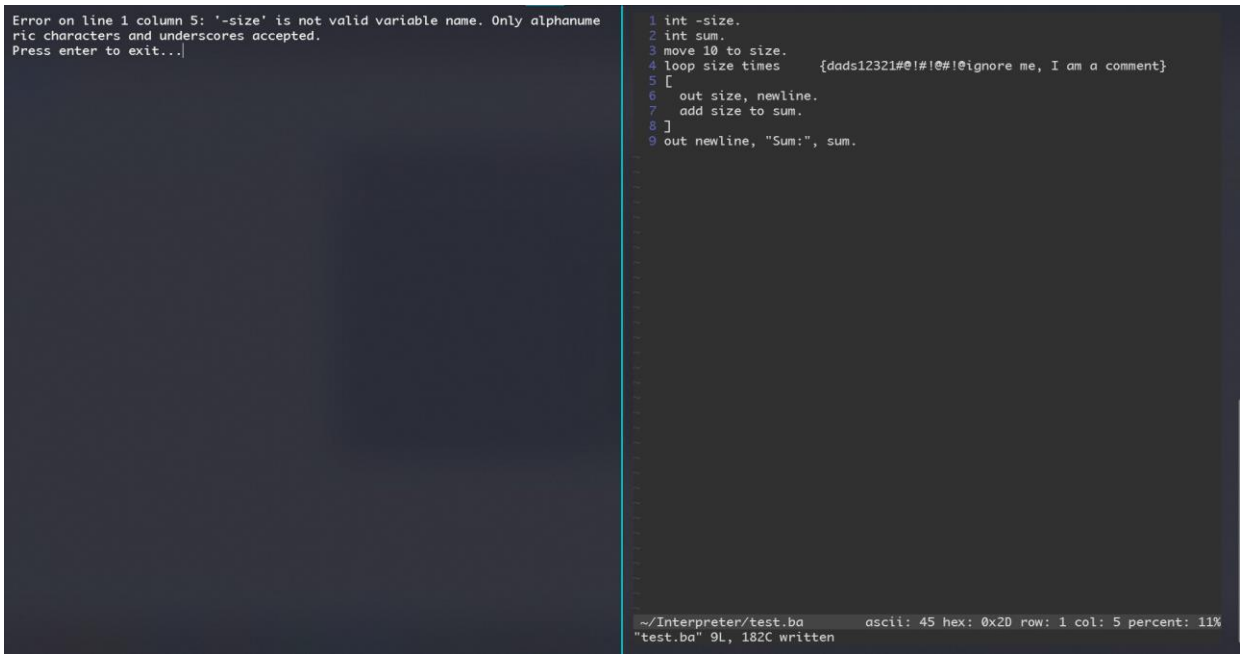
The Identifier Starts With Digit Restriction



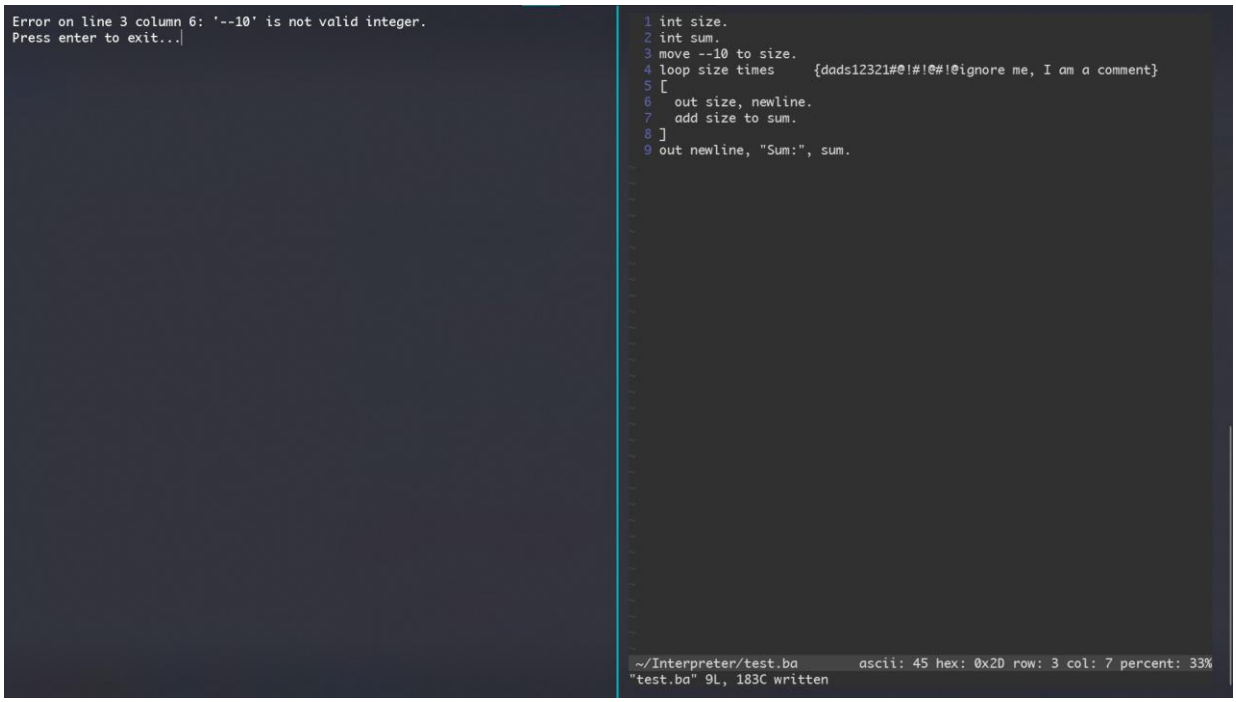
The Identifier Limit Restriction



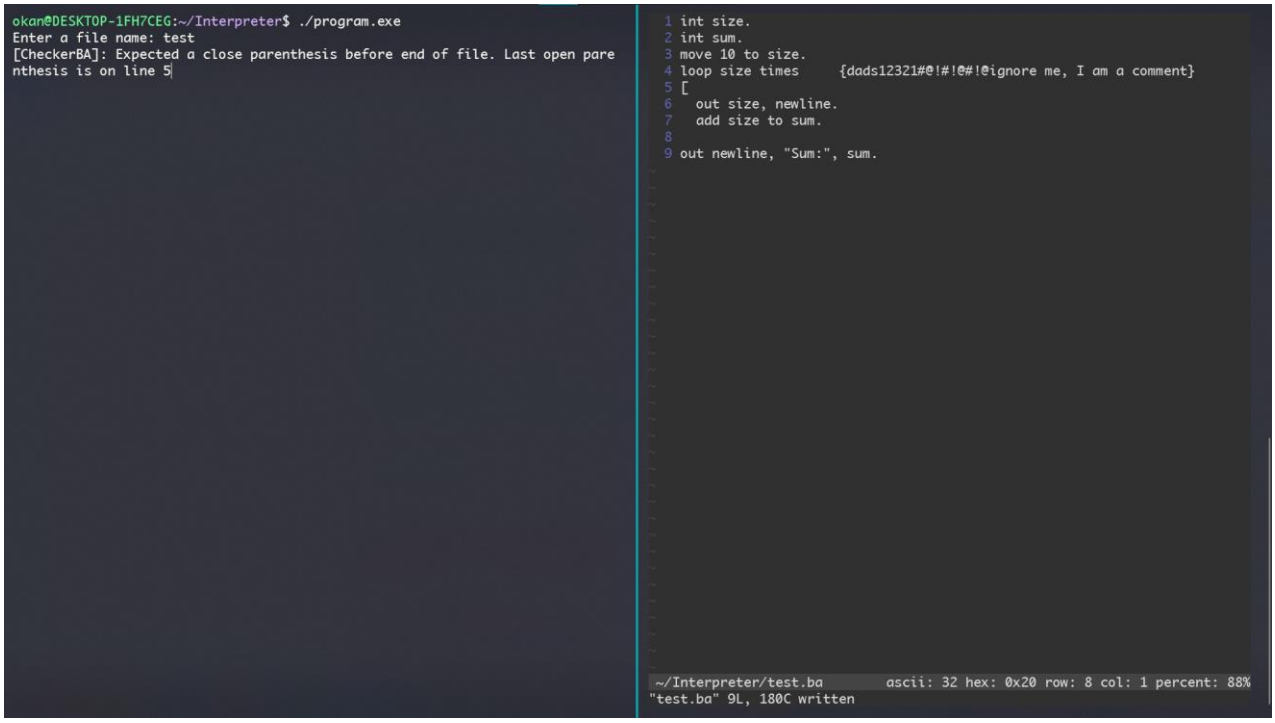
The Identifier Starts With ‘-’ Restriction



Integer contains two or more '-' characters Test

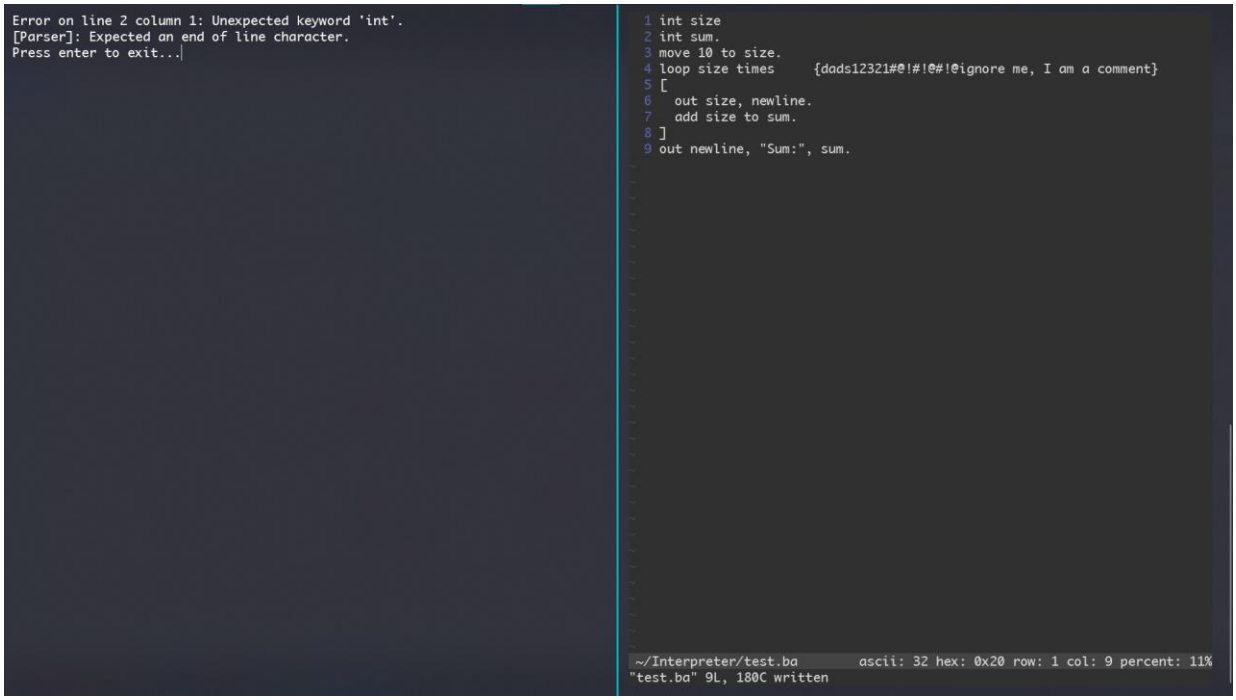


Parenthesis Test

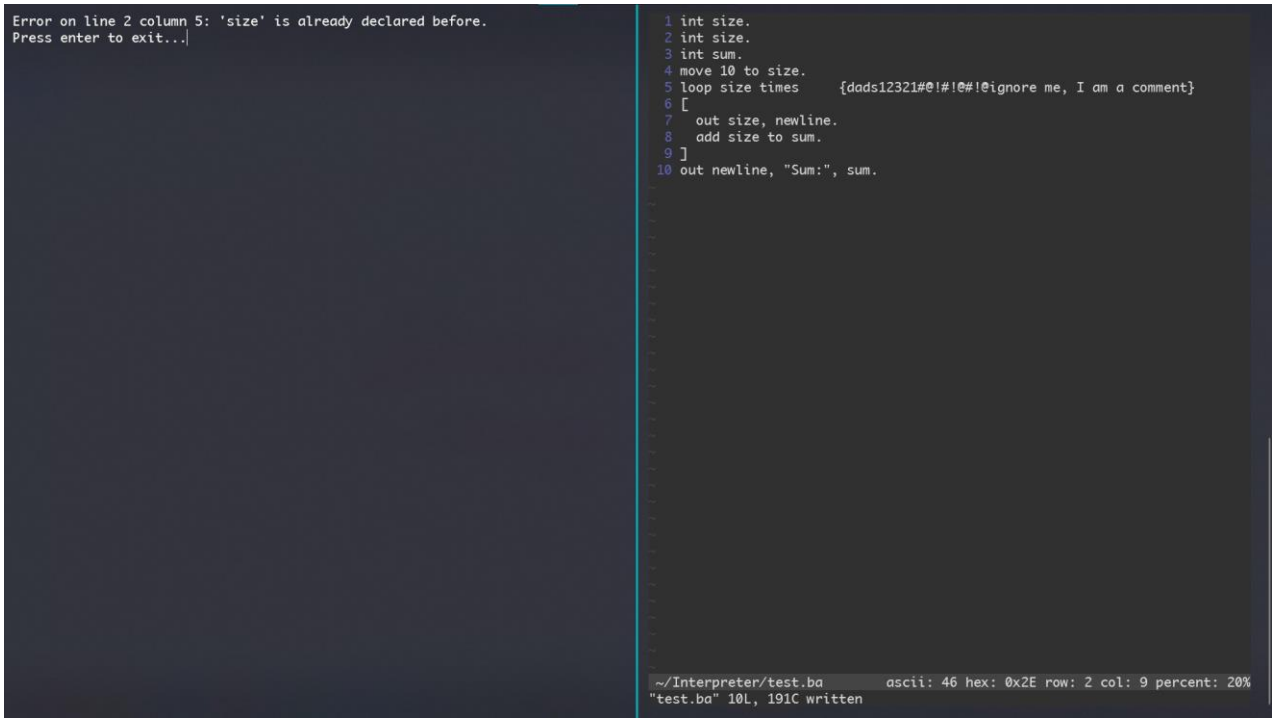


4.c Parser Test

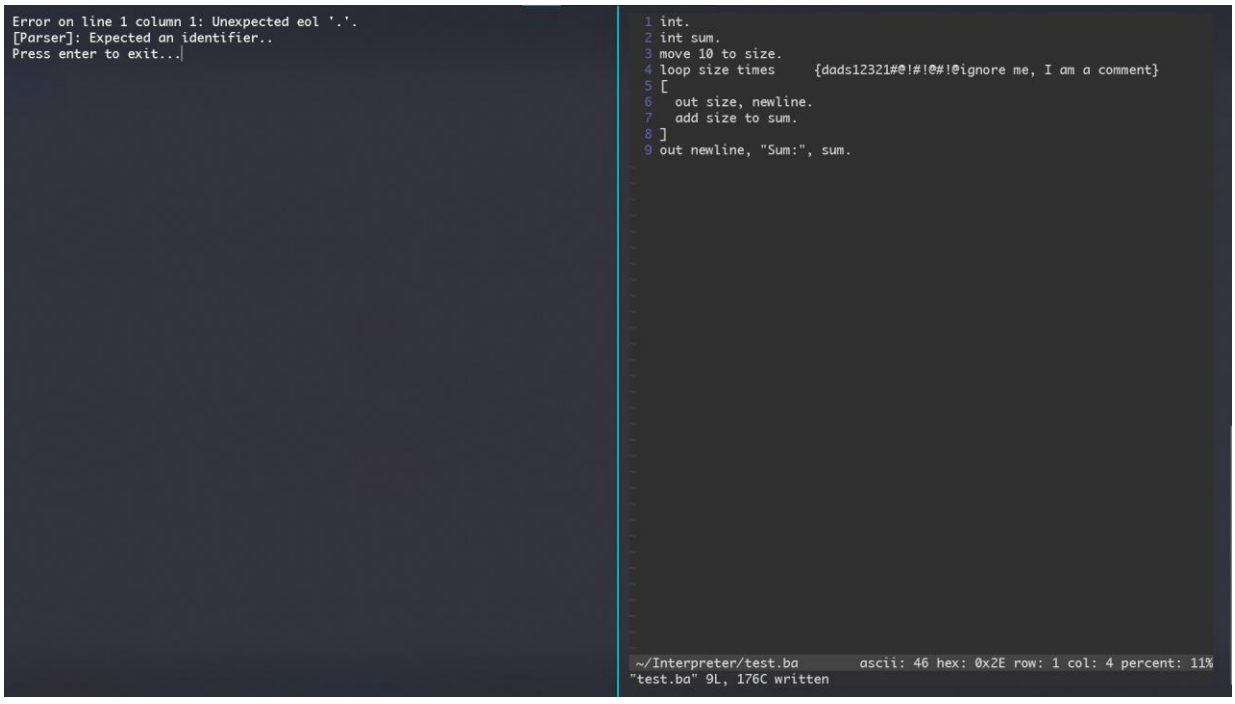
End of line Restriction Test



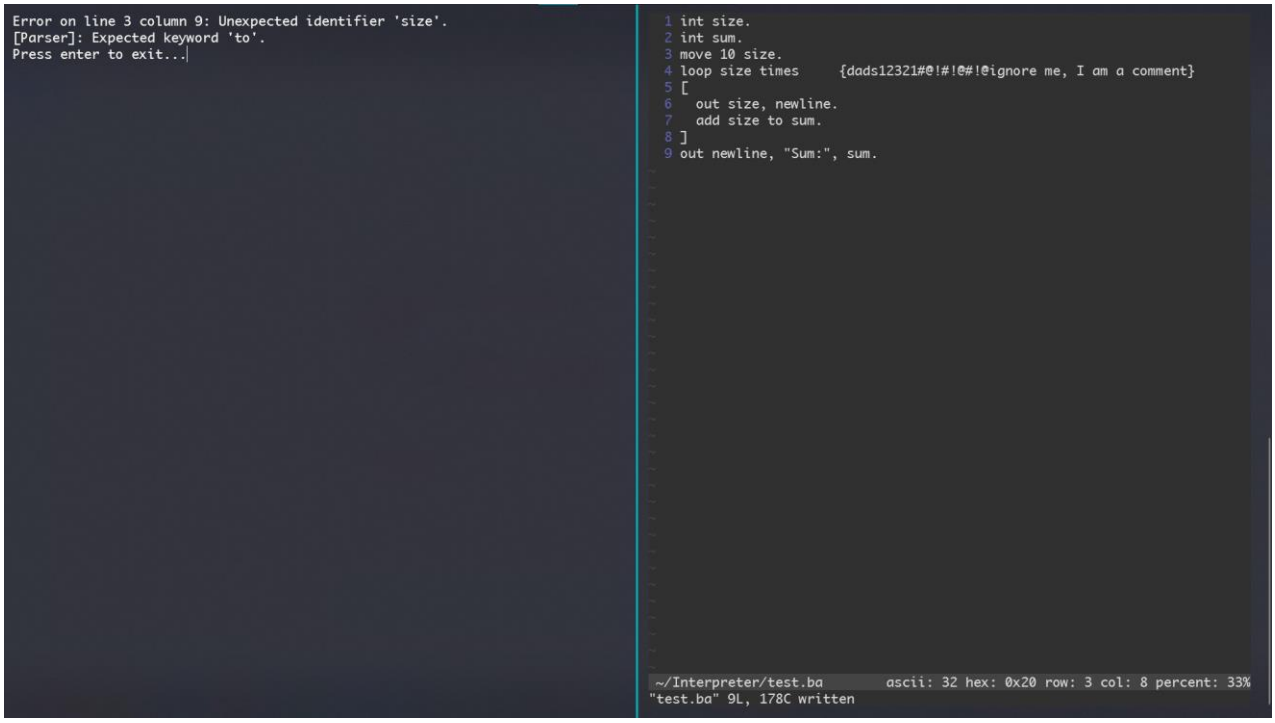
Double Declaration Test



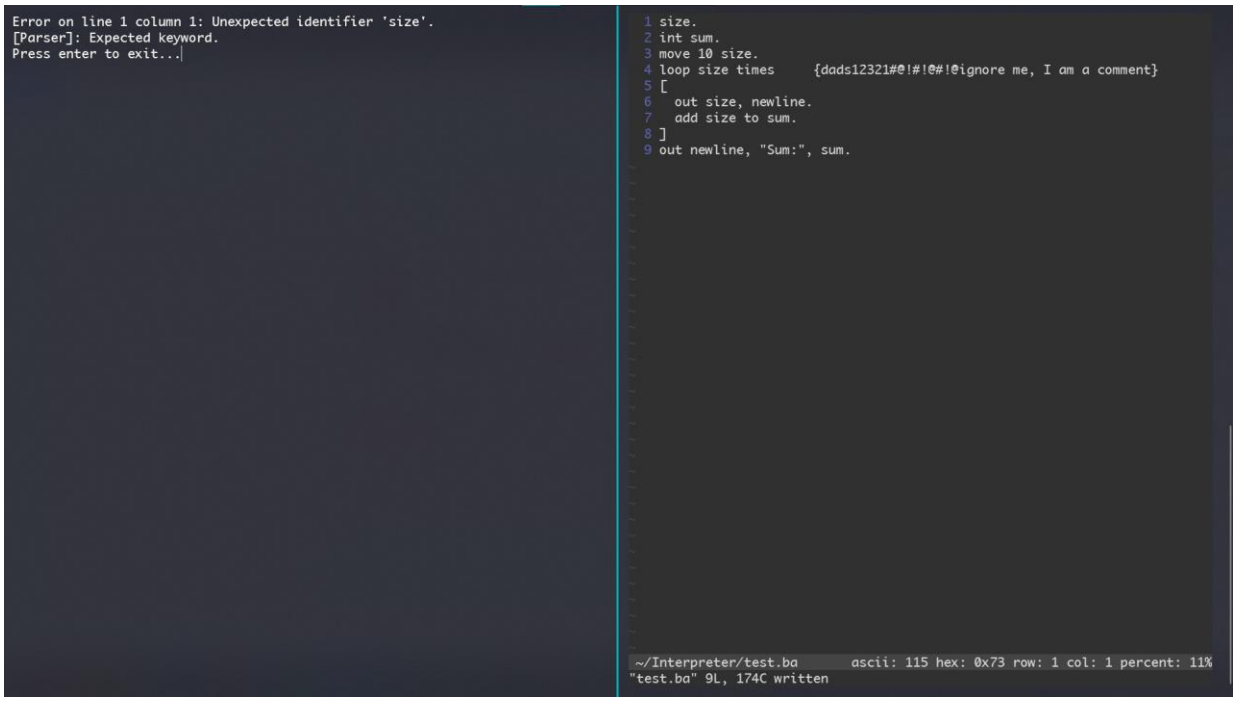
No Identifier Test



The keyword ‘to’ test



Direct Keyword Test



Direct Integer Constant Usage Tests