GROUP ID: 7

PROBLEM STATEMENT: 32

CSN - 352 (COMPILER DESIGN)
CODING PROJECT 1

CHALE INTERPRETER

MEMBERS:
YOUWAN SONI 20114107
UDAY JANGIR 20114102
SOMESHVARI VRAJ 20114093

A BRIEF DESCRIPTION OF THE PROBLEM STATEMENT:

We have to write an interpreter in c/c++ that does addition of integers only.
Including working with lexical analyser, tokens and expressions.

DESCRIPTIONS OF THE ALGORITHM:

Before starting all the assumptions made:

Name of the interpreter: chale interpreter 1.0.1

Valid dataTypes ={INTEGERS}.

Note:
All other types are unacceptable!!

Our algorithm like an regular interpreter reads input from line by line (specifically speaking statement by statement will discuss it later*) and stops as soon as the first error occurs and reports the particular error to the user on the console.

In case of no errors it shows the outputs on the console or does as instructed.

First we are taking an input line from the user as a string then the function lex() is called defined in the **lexer.cpp** file which as it's name suggests performs the lexical analysis by dividing the input string into tokens.

We have made two containers one is our symbol table to keep track of the identifiers that we declared it is a unordered map of type **<string, pair<string, int>>** the first string is basically storing the name of the identifier or its name by which we can access it later and in the pair the string is the basically the type of the identifier and the int in the pair is basically storing the value of the identifier.

The container that we are using for storing the tokens is **deque <pair<string, string>>** So when the lexical analysis is done, to show the output we have declared a function **showOutput** in the **lexer.cpp** file.

Now in the **main.cpp** file we have the following functions:

1. **error(int code)** / **error(int code, string extra_info)**:

Handles different types of error using switch cases. We have defined different error codes for different types of errors to distinguish between them like 11 for unexpected token encounter etc.

These functions are overloaded versions to handle some extra info. if available.

```
// function to show errors
void error(int code) {
    if (!running_state) return;

    switch (code) {

        // if an unexpected token is encountered
        case 11:
            cout << "\033[31mError: Type(Syntax error): Unexpected token " << curr_token.second << "\033[0m" << endl;
            break;

        // if not a valid statement line
        case 12:
            cout << "\033[31mError: Type(Syntax error): Invalid statement line\033[0m" << endl;
            break;

        // if empty expression found
        case 13:
            cout << "\033[31mError: Type(Syntax error): Invalid expression found\033[0m" << endl;
            break;

        // variable declaration error
        case 14:
            cout << "\033[31mError: Type(Syntax error): Variable is not declared properly\033[0m" << endl;
            break;
    }

    // perform a reset
    running_state = false;
    curr_token = make_pair(KEY,NONE);
    tokens.clear();
    return;
}
```

2. **getToken( )**:

Fetches next token from the output of lexer which is stored in ‘tokens deque’ data structure.

```
// function to get the token
pair<string,string> getToken(){
    if (tokens.empty()){
        curr_token = make_pair(KEY,NONE);
        return curr_token;
    }

    pair<string,string> token = tokens.front();
    tokens.pop_front();

    return token;
}
```

3. **match(string token), checkID(string id), matchID( )**:

These functions match the terminals in the grammar rules. For eg. to check the grammar rule **expr_seq -> expression '+' expr_seq** , we have to match '+'. If we find '+' then proceed otherwise show an error.

checkID function look up an id name in the symbol Table.

```
96  // function to match for the terminal
97  void match(string token){
98      if (!running_state) return;
99
100     curr_token = getToken();
101     if (curr_token.second == token) return;
102     else error(21, token);
103 }
104
105 // to check if an ID is in symbol table
106 bool checkID(string id){
107     if (running_state && sym_table.find(id) != sym_table.end()) return true;
108     else return false;
109 }
110
111 // matches ID in the grammar rule and handles errors {11,22}
112 string matchID(){
113     if (!running_state) return "";
114
115     curr_token = getToken();
116     if (curr_token.first == ID) {
117         string id_name = curr_token.second;
118         if (!checkID(id_name)) return id_name;
119
120         // if id_name already in symbol table, handle error
121         else {
122             error(22,id_name);
123             return id_name;
124         }
125     }
126
127     // if some other token found instead of ID
128     else error(11); return "";
129 }
```

4. **expression ( )** :

It checks if the next token is either ID or INT. It is used by another function exprSeq ( ) to make a valid expression for addition. Errors are also handled in this function.

```
// match rule expr -> id | int
int expression(){
    if (!running_state) return INT_MIN;

    // match for expr -> ID rule
    if (curr_token.first == ID) {
        string id_name = curr_token.second;

        if (checkID(id_name)){

            // if variable is declared but not initialised
            if (sym_table[id_name].second == INT_MIN) error(24, id_name);

            else return sym_table[id_name].second;
        }

        // if no such name exists in symbol table
        else error(23,id_name);
        return INT_MIN;
    }

    // match for expr -> INT rule
    else if (curr_token.first == INT) return stoi(curr_token.second);

    // if valid expression not found (i.e. expr -> int | id not matched)
    else {
        tokens.push_front(curr_token);
        error(13);
        return INT_MIN;
    }

}
```

5. **exprSeq ( )** :

Evaluates the expression and returns the answer. Uses recursion to handle multiple expressions. Handles the corresponding errors.

```
// expression sequence, performs addition
int exprSeq(){
    if (!running_state) return INT_MIN;

    curr_token = getToken();
    int result = 0;
    result += expression();
    if (result == INT_MIN) return INT_MIN;

    pair<string,string> next_token = getToken();
    tokens.push_front(next_token);

    // check if sequence ended with ';' else match for '+' instead
    if (next_token.second == SCOLON || next_token.second == COMMA) return result;
    else if (next_token.second == PLUS){
        match(PLUS);
        int remaining_expr = exprSeq();
        if (remaining_expr == INT_MIN) return INT_MIN;
        return result + remaining_expr;
    }
    else return INT_MIN;
}
```

6. updateSymTable (string id, string type, int value), declaration (id), initialisation (id) :

updateSymTable function is used to update the values in the symbolTable.

declaration function handles declaration of variables. i.e if the variable is not initialised, it uses updateSymTable to give that variable value = INT_MIN. It uses recursion to declare multiple variables seperated by comma.

```
187  // update the symbol table
188  void updateSymTable(string id, string type, int value){
189      if (!running_state) return;
190      sym_table[id] = make_pair(type,value);
191      return;
192  }
193
194  bool declaration(vector<pair<string,int>> &ids);
195  bool initialisation(vector<pair<string,int>> &ids);
196
197  // matches declaration rule (assignment -> declaration)
198  bool declaration(vector<pair<string,int>> &ids){
199
200      // if ';' found, declaration ends
201      if (running_state && curr_token.second == SCOLON){
202          while (!ids.empty()){
203
204              // update symbol table for all the ids
205              pair<string,int> id = ids.back();
206              ids.pop_back();
207              updateSymTable(id.first,INT,id.second);
208          }
209          return true;
210      }
211
212      // if multiple declarations using comma, process them
213      else if (running_state && curr_token.second == COMMA){
214
215          // add the declared id the the list of ids
216          string other_id = matchID();
217          ids.push_back(make_pair(other_id,INT_MIN));
218
219          curr_token = getToken();
220          return initialisation(ids) || declaration(ids);
221      }
222      else return false;
223  }
```

initialisation function handles variables that are initialised with some value and updates the symTable with corresponding values. It uses recursion to initialise multiple variables.

Declaration and initialisation work together. i.e some variables can be declared while some can be initialised just by separating them with comma.

```
225  // matches initialisation rule (assignment -> initialisation)
226  bool initialisation(vector<pair<string,int>> &ids){
227      if (running_state && curr_token.second == ASSIGN){
228          int id_value = exprSeq();
229          if (id_value == INT_MIN) return false;
230
231          // add the id with its initialisation value to list of ids
232          pair<string,int> id = ids.back();
233          ids.pop_back();
234          id.second = id_value;
235          ids.push_back(id);
236
237          // handle multiple initialisations by checking comma
238          curr_token = getToken();
239          if (curr_token.second == COMMA){
240
241              // update ids list with new id
242              string other_id = matchID();
243              ids.push_back(make_pair(other_id,INT_MIN));
244              curr_token = getToken();
245
246              return initialisation(ids) || declaration(ids);
247          }
248          else tokens.push_front(curr_token);
249
250          // if semicolon found, then assign values to all ids in list
251          match(SCOLON);
252          while (!ids.empty()){
253              pair<string,int> id = ids.back();
254              ids.pop_back();
255              updateSymTable(id.first,INT,id.second);
256          }
257          return true;
258      }
259      else return false;
260  }
```

7. **assignment ( )** :

Used to check for grammar rule assignment -> declaration | initialisation | reassign.

Uses functions declaration and initialisation.

It can also reassign values to already defined variables by updating their values in the symbolTable.

```
// to handle assignment -> declaration | initialisation | reassign
bool assignment(){
    if (!running_state) return false;

    // if 'int' keyword found, we should declare or initialise new variable
    if (curr_token.second == INT){

        // match ID tokens and handle errors
        string id_name = matchID();
        vector<pair<string,int>> ids ;
        ids.push_back(make_pair(id_name,INT_MIN));

        curr_token = getToken();
        if (declaration(ids) || initialisation(ids)) return true;

        // if not a proper assignment syntax show error
        else{
            error(14);
            return false;
        }
    }

    // otherwise reassign some new value to the variable
    else if (curr_token.first == ID){
        string id_name = curr_token.second;
        if (checkID(id_name)) {
            match(ASSIGN);
            int id_value = exprSeq();
            if (id_value == INT_MIN) return false;
            match(SCOLON);
            updateSymTable(id_name,INT,id_value);
        }
        else {
            error(23, id_name);
        }
        return true;
    }
    else return false;
}
```

8. **print ( ) , exitProgram ( )** :

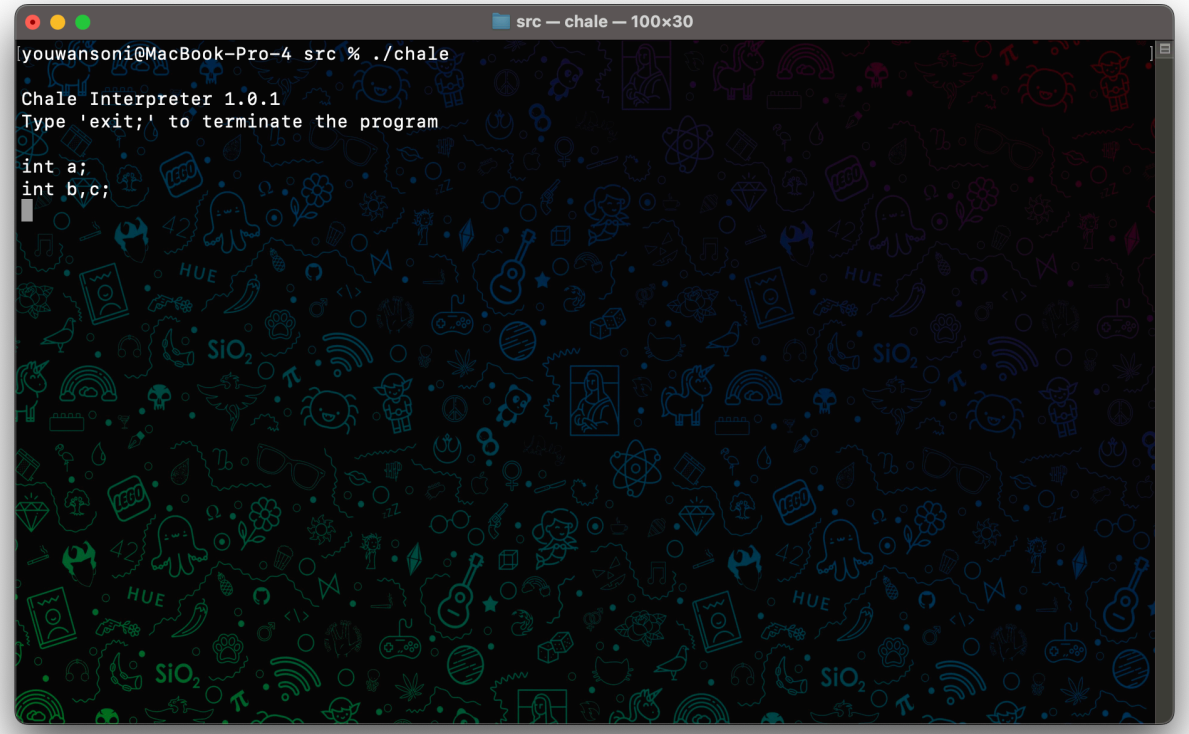
print function uses cout keyword to show the output of an expression. It checks for **“<<“** token and shows error if any other token found.

exitProgram function finds the keyword “exit” and terminates interpreter program in case its found, otherwise keep taking input lines.

```
302  // prints using 'cout' keyword
303  bool print(){
304      if (running_state && curr_token.second == COUT){
305          match(DLT);
306          int value = exprSeq();
307          match(SCOLON);
308          if (value == INT_MIN) return false;
309
310          // prints only if we have not encountered any error
311          if (running_state) cout << value << endl;
312          return true;
313      }
314      else return false;
315  }
316
317  // exit the interpreter using 'exit' keyword
318  bool exitProgram(){
319      if (running_state && curr_token.second == EXIT) {
320          match(SCOLON);
321          if (running_state) exit(0);
322          else return false;
323      }
324      else return false;
325  }
```

TEST CASES AND DEMONSTRATION

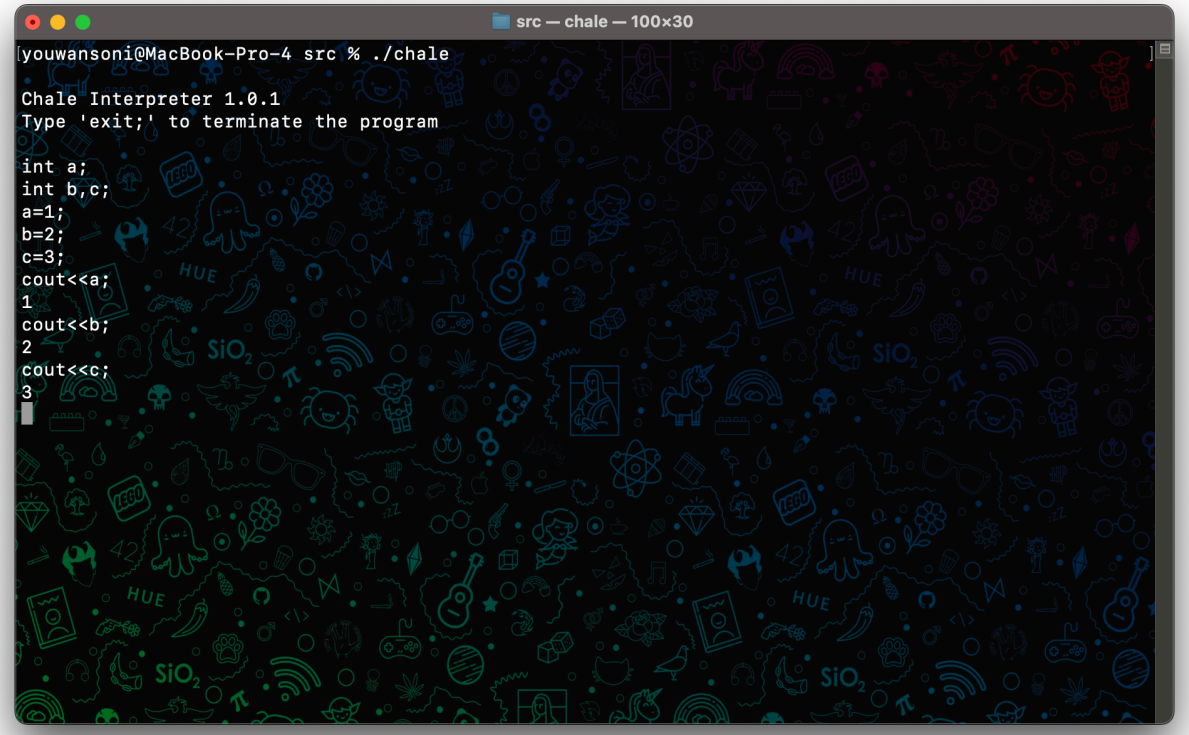
1. **Declaring some variables not initialising them like shown:**



In the first line we are declaring a variable **a** of type int.

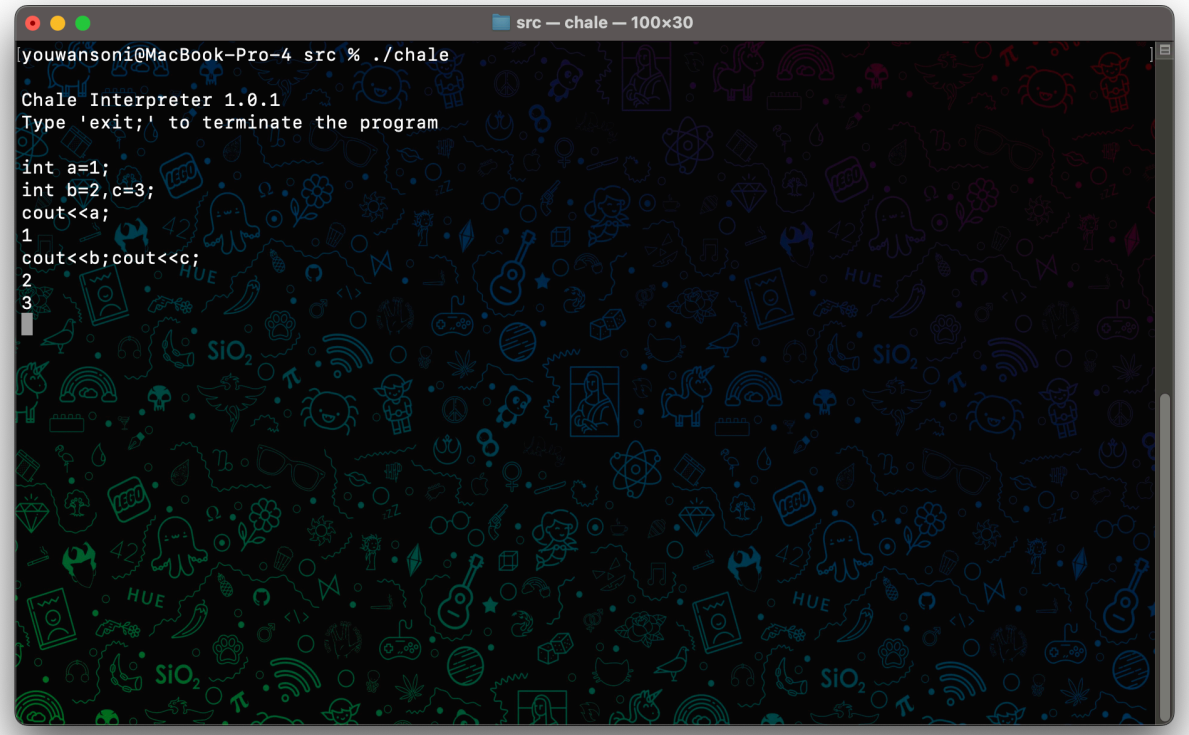
In the second line we are declaring two variables **b** and **c** both of type **int** and in a similar way we can declare more than 2 variables also.

2. Initialising the already declared variables, creating new variables and re-assigning different values to variables.



In the first two lines we've declared 3 variables and in the next 3 lines we have initialised them.

Now we are printing them to make sure everything is stored correctly.

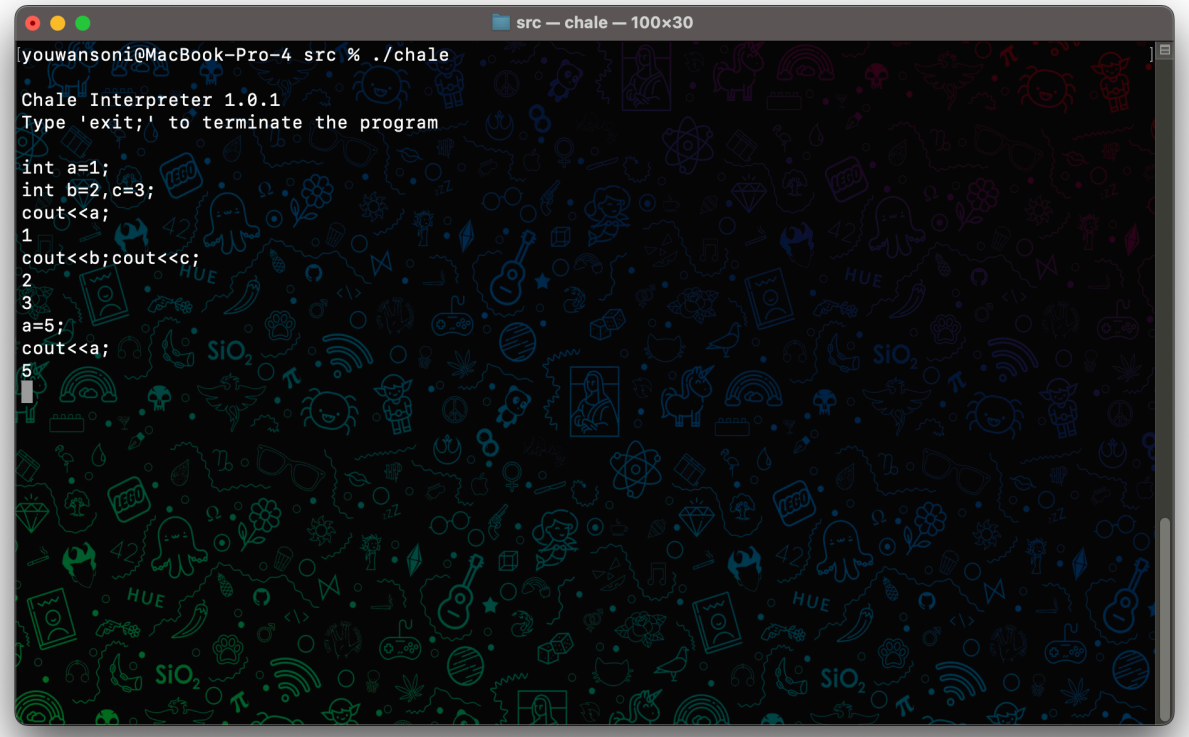


The above demonstration is a test case of first declaration and then initialisation

But done only by one statement.

In the first two lines we initialised 3 variables with different values.

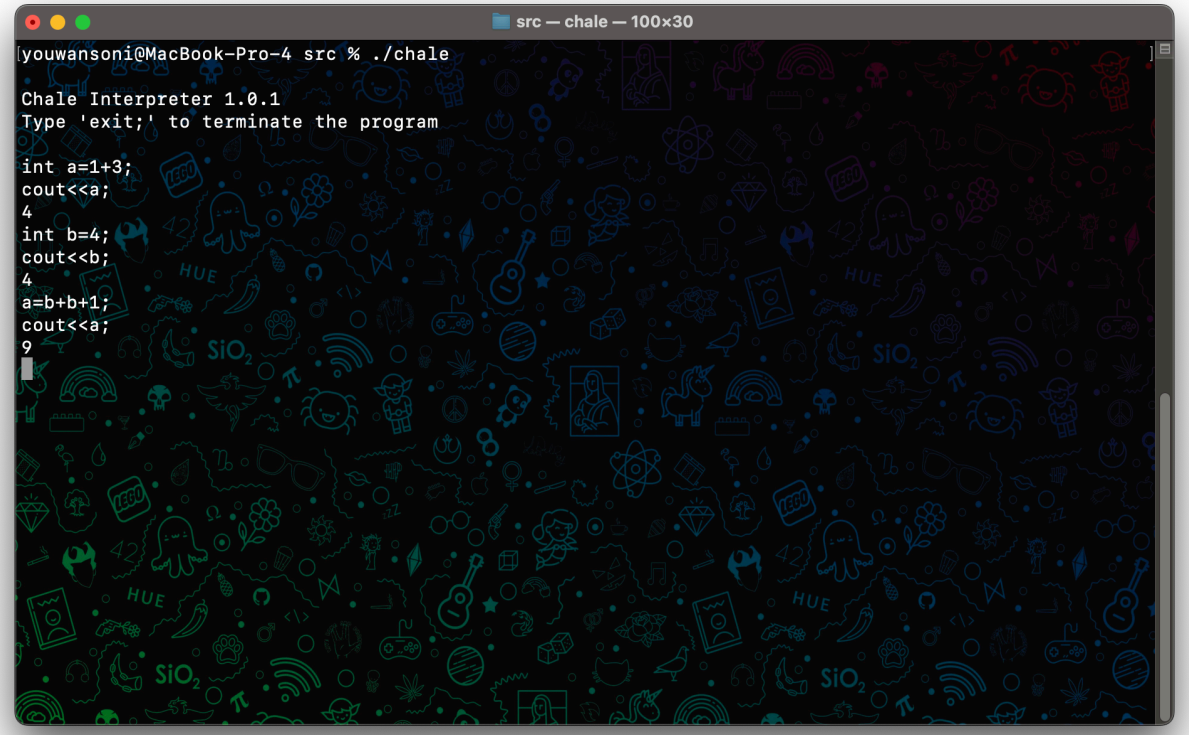
For making sure we are then printing the values of these identifiers which comes out to be correct.



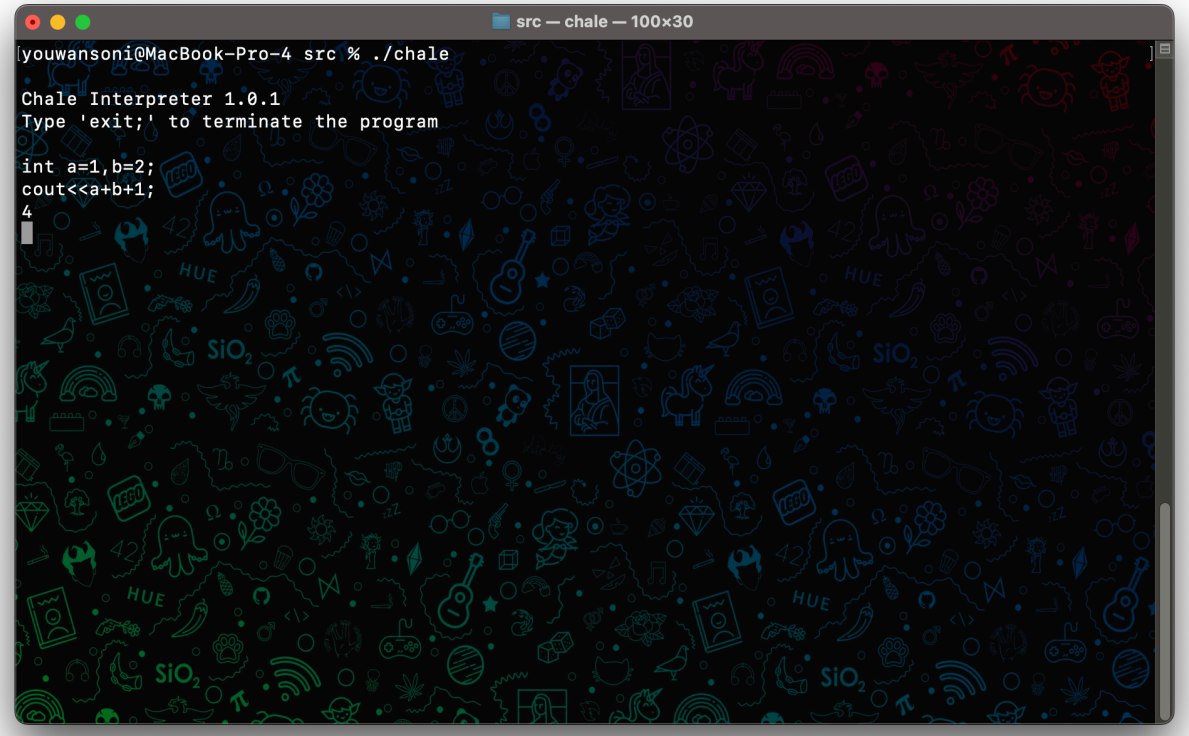
This above test case shows the reassignment of a already declared identifier(**a**).

In the last second line we are reassigning the value of **a** with the value 5.

3. Dealing with expressions.

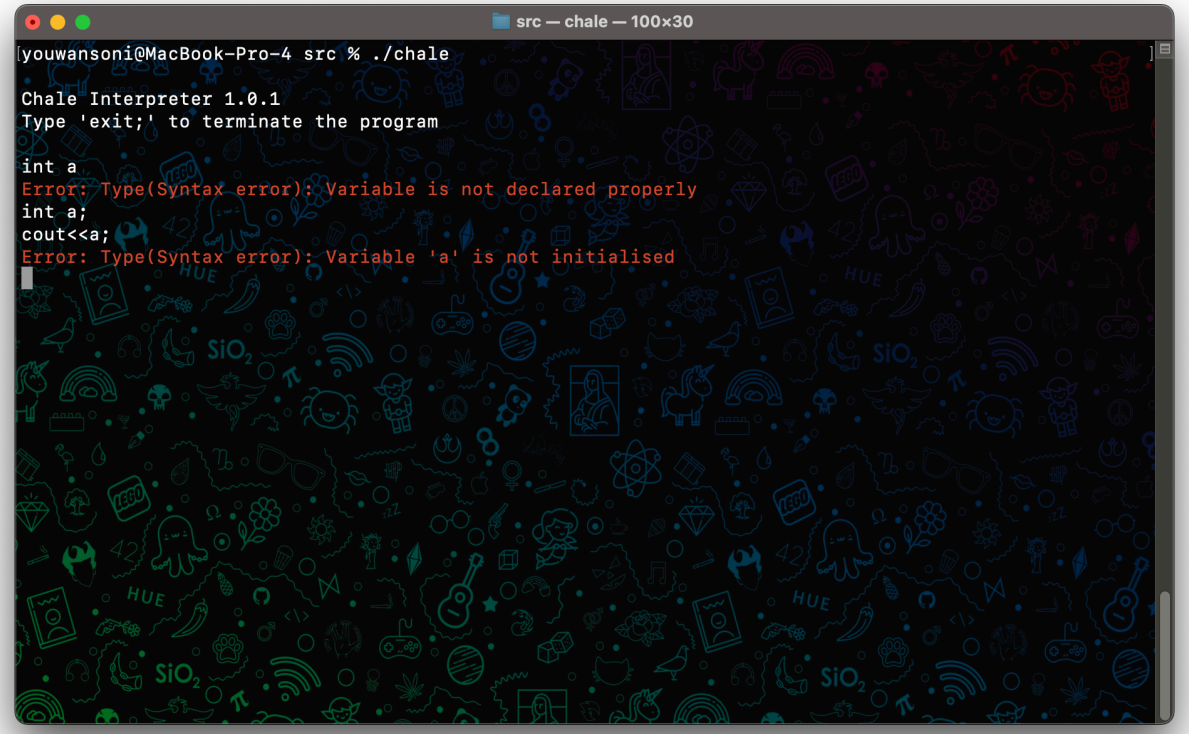


In the first line we are declaring the variable **a** with the value of the expression **1+3**. Then we are reassigning the value of **a** with the value of expression **b+b+1**.



Now in this screenshot we are demonstrating the expression being used in **cout** statements. here we are printing the value of the expression **a+b+1**.

4. Errors

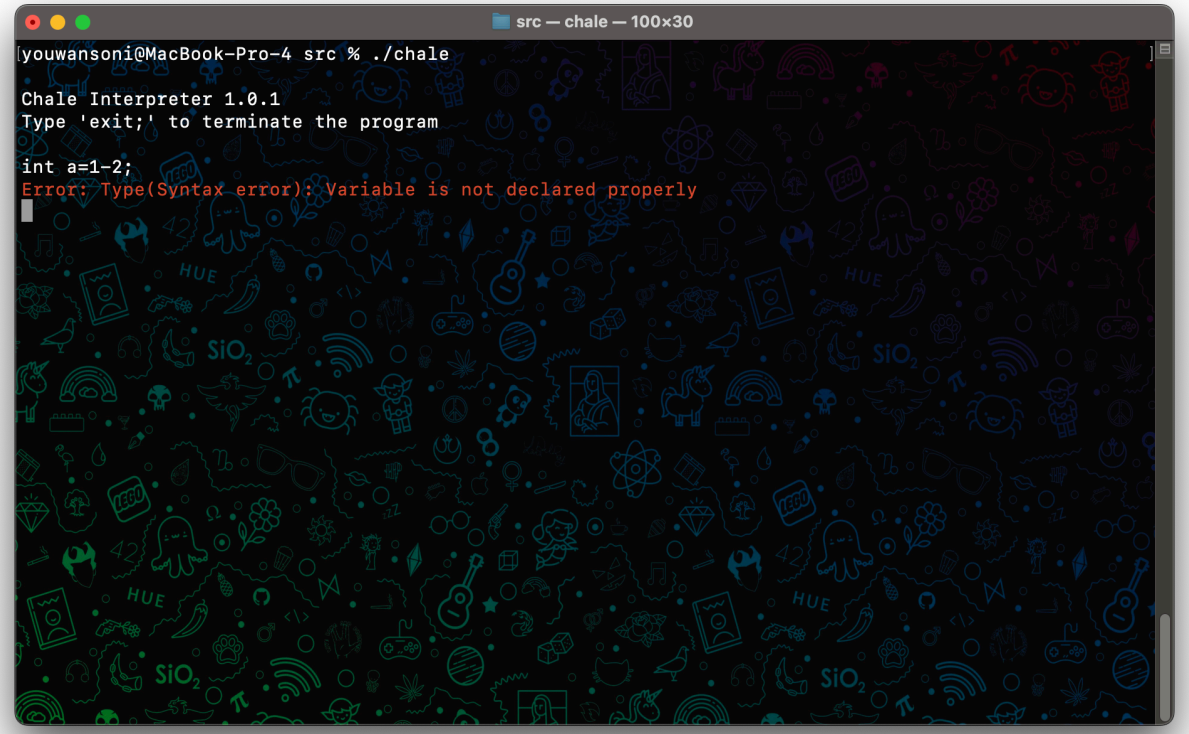


Here we are committing two types of syntax errors.

In the first line we are declaring in the wrong way.

Now **in the second line** we only declared the variable **a**.

Now we are committing another **error** by trying to print an identifier which is not initialised.



This above screenshot demonstrates how it only validates expression that are valid by **syntax** and **only perform addition** nothing else.