REPORT
112001042

FILES

CHANGELOG.md : contains all the updates i have done
INSIDE compilercpy
calc3.h defines all the structures (inside include)
Inside src

  compiler.l
  compiler.y
  Compiler.c
  toc.c
  syntaxTree.c

  Testcases {test}

calc3.h

Defines all the types
The first typedef statement defines an enumeration called nodeEnum.This enumeration is used to specify the type of a node in an abstract syntax tree.

The second typedef statement defines another enumeration called symTabEnum.This enumeration is used to specify the type of a symbol in a symbol table.

The next several struct statements define the structures for three different types of nodes in an abstract syntax tree: constants (conNodeType), identifiers (idNodeType), and operators (oprNodeType). Each of these structures contains one or more fields that hold relevant information for that type of node.

The nodeTypeTag structure is used to represent a node in the abstract syntax tree. Depending on the type of node, the union field will contain either a conNodeType, an idNodeType, or an oprNodeType.

The sym structure represents a symbol in a symbol table.

compiler.c

The ex() function takes a pointer to a node as input, and recursively evaluates the AST by performing the appropriate operations based on the type of the node.

In addition to the basic arithmetic and logical operators, the code also includes operations for printing values (PRINT, PRINT_List), declaring variables (DECLARE, DECLARE_List, DECLARE_Fn), calling functions (CALL), and control flow statements (IF, WHILE, Main, etc.).

Helper functions like getVal(),updateVal() etc are also written in this c file.
PrintSyntaxTree prints the AST.

compiler.l

This code is a Lex file, which is used to generate a lexical analyzer for a given language. The code defines a set of rules that describe the tokens that can be recognized by the analyzer. Each rule consists of a regular expression pattern and an action to be taken when that pattern is matched.

compiler.y

This contains the grammar for the language and some functions to help create nodes of different types.

IF-ELSE , While , main , operations, Print ,variable declarations,assignment
work.

toc.c

Containts the code to convert the ast to c code

LEXICAL ANALYSIS

Lexical analysis is the first phase of compiler design that involves converting a sequence of characters into a sequence of tokens. The process involves breaking down the input source code into individual lexemes and then identifying the corresponding tokens. The tokens are then passed onto the next stage of the compiler for further processing.

In compiler.l regular expressions like print,decl,if, else,while etc are converted into tokens .
Digits are converted into token NUMBER

SYNTAX ANALYSIS

Syntax analysis, also known as parsing, is the second stage of the compiler front-end. It takes the sequence of tokens generated by the lexical analysis stage and checks whether the program follows the grammar rules of the programming language.

The grammar rules define the valid syntax of the programming language, including rules for defining variables, operators, functions, and control structures. The syntax analysis stage checks the sequence of tokens to determine whether they form a valid sentence in the programming language.

Parser constructs the syntax tree (this happens in compiler.y)

Evaluating syntax tree

In compiler.c , the syntax tree is traversed and appropriate operations are done.
Syntax tree is also printed. In the next stages of the project , from the syntax tree C code will also be generated.

My implementation

<u>Global declaration:</u>

Yacc rules decl_stmt and varList is used for global declarations
varList can be a single variable (variable or array) or a list of variables(used a recursive type declaration here) . Declare statements are converted into DECLARE node

Syntax

decl <var_names> ; enddecl

Declarations are implemented using different cases in c file, different for arrays, list of variables or a single variable.

<u>Relational operations</u>

Using yacc rules expr +,-,/,*,%,<,>, logical operators are implemented. This was from previous labs

<u>Main function:</u>

There is a main block , which for now can have return type only integer.
Syntax:
integer main() {

begin
<code>
end
}

The code is converted into nodes and evaluated directly using ex() function in compiler.c

<u>Conditional statements</u>

Inside rule stmt, rules for different types of conditional statements are mentioned.
If-else:
An if node is created and evaluated
Syntax :
if <expr> then <stmnts>
else
<stmnt>
endif ;
While evaluating if node, first the expression is evaluated and if it is true then the statements in the then block is evaluated or the else block is evaluated.
While works the similar way

Syntax:
while <expr> do
<code>
endwhile ;

<u>Recursion</u>
When a function is called, a new local symbol table (saveTab) is created for that function. This local symbol table is used to store all the local variables and their values used within the function. This local symbol table is separate from the global symbol table, which stores global variables and their values.

<u>C code printing</u>
Once the syntax tree is generated and the nodes have been created, it is possible to recursively traverse the syntax tree and generate C code that corresponds to the input source code.

To run:

cd compilercpy
<u>make test</u> —> gives c code in the terminal
(Give the testcase in test tile and run make test)
<u>make output</u> —> prints the c code in bin/output.c

make testc runs the c code and prints the output in the terminal