PROJECT REPORT

NAME:- B.VISHAL REDDY
ROLLNO:201501173

Scanner.l:- Token declarations
Parser.y:- Bison Grammar
Classes.h:- Class declarations
Methods.cpp:- Method definitions declared in classes.h
How to Execute :-

Step1: make
Step2: ./bcc testfile 2>llout
Step3: Test the normal interpreter
Step4: llvm-as llout -o llbc
Step5: lli llbc
Step6:Test the llvm interpreter

1 .FLAT B LANGUAGE DESCRIPTION:

This language consists of two blocks :- declblock and codeblock.
Declblock :This block contains all variable declarations.
Codeblock: This block contains all statements.

All the variables have to be declared in the declblock{....} before being used in the codeblock{...}. Multiple variables can be declared in the statement and each declaration statement ends with a semi-colon.

SPECIFICATIONS:

1. Data Types

Integers and Array of Integers.
int data, array[100];
int sum;

2. Expressions

Arithmetic Expressions (+,-,*,/,%)
Boolean Expressions (&&,||,^,!=,==,&,|)

3. For loop

```
for i = initial value, limit value {
/*here incremented value is 1*/
   statements;
}

for i = initial value, limit, incrementation {
   statements;
}
```

4. if-else statement

```
   if Boolean expression {
   statements;
   }
   ....

   if Boolean expression {
   statements;
   }
   else {
   Statements;
   }
```

5. while statement

```
   while Boolean expression {
        statements;
   }
```

6. conditional and unconditional goto

```
   got label ;
   goto label if boolean expression;
```

7. print/read

```
    print "blah...blah", val;
    println "new line at the end";
    read sum;
    read data[i];
```

2. SYNTAX ANALYSIS

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language specified in bison and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.
Bison constructs LALR(1) parser tables by default.

Example:

```
Program:DBLOCK '{' Declarations '}' CBLOCK '{' Statements '}' {$$ = new Prog($3,$7);}
Declarations: {$$ = new Decls();}
    | Declarations Declaration SC {$$->push_back($2);}
Declaration:
    TYPE Variables {$$ = new Decl(string($1),$2);}
Variables:
    Variable {$$ = new Vars();$$->push_back($1);}
    | Variables COMMA  Variable { $$->push_back($3); }
Variable:
    ID { $$ = new Var(string("Normal"),string($1));}
    | ID '[' INTEGER ']' { $$ = new Var(string("Array"),string($1),$3);}
```

The above grammar is for declblock.
Capital letter names are tokens declared in scanner.l and it will look like <int,TYPE> .
During syntax analysis if input string cannot be produced by the syntax tree it throws an error saying syntax error at line number.
Example:

```
declblock{
int a,b
}
Output :-   Error:syntax error at line number 2
```

3. DESIGN OF AST

My Ast design is done using OOPS.Class heirarchy for my ast design is as follows:-

```
Class astNode{
};

Class Decls::public astNode{
};
Class Decl::public astNode{
};
Class Variables::public astNode{
};
Class Variable::public astNode{
};
Class stmts::public astNode{
};
Class Expr::public astNode{
};
Class Arthexpr::public Expr{
};
Class Boolexpr::public Expr{
};
Class Assignment::public stmt{
};
Class ifstmt::public stmt{
};
Class whilestmt::public stmt{
};
```

4. SEMANTIC ANALYSIS

Semantic analysis is done after AST construction.Now we traverse the tree and analyse the following semantics:-
1.Check if the variable used in codeblock is declared in declblock.
2.Check if array accessed is not out of bound.
3.Check for variable redeclaration.
4.Check for goto label redeclaration.
5.Check if goto label is declared or not.

Variables and their type,length etc is stored in the symbol table for checking.
Traversing is done using visitor design pattern.

5.VISITOR DESIGN PATTERN

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
The implementation proceeds as follows. Created a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.
The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class.

Example:

```
class Visitor
{
  public:
            virtual void visit(Decls*)=0;
            virtual void visit(Decl*)=0;
            virtual void visit(Vars*)=0;
            virtual void visit(Var*)=0;
}
```

```
class Traverse:public Visitor{
public:
    void visit(Decls* p);
    void visit(Decl* p);
    void visit(Vars* p);
    void visit(Var* p);
}
void Traverse::visit(Var* p){
  if(p->declType.compare("Array") == 0){
         table.insert(p->declType,p->name,p->dataType,p->length,0);
  }
  else
         table.insert(p->declType,p->name,p->dataType,1,0);
}
```

Semantic analysis and interpreter is done using visitor design pattern (2 times traversing the tree).

6.DESIGN OF INTERPRETER

Interpreter is done using visitor design pattern.
For assigning values to variables a field is used for values in symbol table.If Value is updated then it is updated in symbol table.

Example:

```
int Evaluate::visit(Assignment* p){
if(!p->loc->is_array()){
  for(int i=0;i<table.symlist.size();i++){
         if(table.symlist[i]->name==p->loc->getVar()){
         table.symlist[i]->arr[0] = p->expr->accept(v);
         }
         }
  }
  if(p->loc->is_array()){
         for(int i=0;i<table.symlist.size();i++){
         if(table.symlist[i]->name==p->loc->getVar()){
         table.symlist[i]->arr[p->loc->getExpr()->accept(v)] = p->expr->accept(v);
         }
         }
  }
  return 1;
}
```
In the above example for assignment statement symbol table finds lhs variable and update the rhs value in the symbol table.

7.DESIGN OF LLVM CODE GENERATOR

An LLVM module class is the top-level container for all other LLVM IR objects. An LLVM module class can contain a list of global variables, functions, other modules on which this module depends, symbol tables, and more. Here's the constructor for an LLVM module:

```
static Module *MyModule = new Module("FlatB compiler",getGlobalContext());
```
The next important class is the one that actually provides the API to create LLVM instructions and insert them into basic blocks: the IRBuilder class.

```
static IRBuilder<> Builder(Context);
```
When the LLVM object model is ready, you can dump its contents by calling the module's dump method.

All Variables are declared globally.Now in Code block main function is created for the basic blocks.A basic block is created before the first statement ,before a Statement that immediately follows a conditional or unconditional goto ,before a Statement that is the target of a conditional or unconditional goto,before if stmts,before else stmts,after if or else merge stmts,before while stmts,after while stmts,before for stmts,after for stmts.

Example:
```
declblock{
        int a;
}
codeblock{
        a=2;
}
```
LLVM assembly code generated is :-
```
; ModuleID = 'FlatB compiler'

@a = global i32 0, align 4
```

```
define void @main() {
entrypoint:
  store i32 2, i32* @a
  ret void
}
```

After the LLVM code generated using llvm-assembler this is converted to bit code format.
We use lli to interpret the code generated.
Using the llc we can convert LLVM bitcode to the target assembly code and if target architecture is not specified it uses architecture of host machine.

PERFORMANCE COMPARISION

With intel core i5 processor :

1. Wall clock time comparision :-

Program	Testcases	Normal	lli	llc
Bubblesort	100000	8.227s	0.461s	0.379s
GcdLcm	100000	0.885s	0.660s	0.578s
NcR	100000	1.151s	0.076s	0.047s

Huge variation in bubblesort time for normal interpreter is because of cout used in the interpreter for printing which is very slow compared to lli,llc print.

Here wall clock time used depends on processor used.
With i7 processor time taken will be lower than the above results.

2. Number of Instructions :-

Program	Normal	lli	llc
Bubblesort	46839549079	1546518946	1449782331
GcdLcm	2267931150	858297306	817140733
NcR	4903952844	215697438	183102399