Compilers
Project Report

Harshit Mahajan
201501231

The aim of the project was to write a compiler for the FlatB Programming language. FlatB is a simple imperative language similar to C.

The compiler construction was done in 3 phases.

- **Phase 1 :**
  Writing a parser for parsing the source code, using flex and bison, and detecting any errors.
- **Phase 2 :**
  - **a)** Constructing an AST of the given source code, using bison, defining a custom class for each type of node. This is done using Visitor Design Pattern.
  - Interpreting the AST generated, using Visitor Design Pattern.
- **Phase 3 :**
  Generating IR code from each of the nodes in the AST, using LLVM. I also did Performance Comparison using my Interpreter, lli and llc on 3 benchmark problems : bubblesort.b, factorial.b and cumulative.b.

1) FlatB Programming Language Description :-

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 semicolon.

A. Expressions:

a. Arithmetic Expression : - Addition, Subtraction, Multiplication, Division, Modulus
b. Boolean Expression : - <,>,<=,>=,==,!= are supported.

B. if-else statement

```
 if expression {
        ....
  }

  if expression {
        ...
  }
 else {
        ....
 }
```

C. for loop

```
for i = 1, 100 {
                 .....
}
for i = 1, 100, 2 {
                 .....
}
```

D. while statement

```
   while expression {

   }
```

E. Conditional and Unconditional Goto

```
        goto label;
        goto label if expression;
```

F. Print

```
print "blah...blah", val;
println "new line at the end";
```

G. Read

```
read sum;
read data[i];
```

2) Syntax and Semantics

```
program             :       DECL_BLOCK '{' declaration_list '}'  CODE_BLOCK
'{' statement_list '}'


/* ------------ decl_block starts   ---------------- */

declaration_list  :         /* epsilon */
                    |       declaration_list single_line ';'

single_line         :       TYPE variables

variables           :       variable
                    |       variables ',' variable

variable            :       IDENTIFIER
                    |       IDENTIFIER '[' NUMBER ']'
                    |       IDENTIFIER '=' NUMBER

/* code_block starts */

statement_list      :       /* epsilon */
                    |       statement_list IDENTIFIER ':' statement
                    |       statement_list  statement

statement           :       assign_expr        ';'
                    |       if_statement
                    |       while_statement
                    |       for_statement
                    |       goto_statement   ';'
                    |       print_statement  ';'
                    |       read_statement   ';'

assign_expr         :       IDENTIFIER               '=' expr
                    |       IDENTIFIER '[' terminal ']' '=' expr

terminal            :       IDENTIFIER
                    |       NUMBER
```

```
expr            :       terminal
                        |       IDENTIFIER '[' terminal ']'
                        |       arith_expr

arith_expr              :       expr '+' expr
                        |       expr '-' expr
                        |       expr '/' expr
                        |       expr '*' expr
                        |       expr '%' expr

bool_op                 :       EQUAL_EQUAL
                        |       GT_EQUAL
                        |       LT_EQUAL
                        |       NOT_EQUAL
                        |       '>'
                        |       '<'

bool_expr               :       expr bool_op expr
                        |       bool_expr OR bool_expr
                        |       bool_expr AND bool_expr

if_statement            :       IF bool_expr '{' statement_list '}'
                        |       IF bool_expr '{' statement_list '}' ELSE '{'
statement_list '}'

goto_statement          :       GOTO IDENTIFIER
                        |       GOTO IDENTIFIER IF bool_expr

while_statement          :      WHILE bool_expr '{' statement_list '}'

for_statement           :       FOR assign_expr ',' terminal '{' statement_list '}'

                        |       FOR assign_expr ',' terminal ',' terminal '{'
statement_list '}'

read_statement          :       READ terminal
                        |       READ IDENTIFIER '[' terminal ']'
```

```
print_statement  :      PRINT   contents  { $2->line = false; $$=$2;}
                 |      PRINTLN contents  { $2->line = true; $$=$2;}

contents         :      content
                 |      contents ',' content

content          :      STRING_LITERAL
                 |      IDENTIFIER
                 |      IDENTIFIER '[' terminal ']'
/*------------ Terminal Symbols -------- */
IF
FOR
WHILE
ELSE
BREAK
CONTINUE
RETURN
AND
OR
DECL_BLOCK
CODE_BLOCK
TYPE
NUMBER
IDENTIFIER
ETOK
EQUAL_EQUAL
STRING_LITERAL
PRINT
PRINTLN
READ
LABEL
```

3) Design of AST

I designed the AST such that all the rules are mentiones alongside the grammer in **parser.y**. All the classes for non-terminals were declared ClassDefs.h and required functions and variables were defined Classes.cpp.

Classes Declared For Genrating AST:-

```
class BaseAst
class Variable:public BaseAst
class Variables:public BaseAst
class DeclList:public BaseAst
class Statement:public BaseAst
class StatementList:public BaseAst
class Terminal:public BaseAst
class Expr:public BaseAst
class ArithExpr:public BaseAst
class AssignExpr:public Statement
class BoolExpr:public BaseAst
class IfStmt:public Statement
class WhileStmt:public Statement
class GotoStmt:public Statement
class ForStmt:public Statement
class ReadStmt:public Statement
class PrintStmt:public Statement
class Prog:public BaseAst
```

4) Visitor Design Pattern and how it is used.

I have used visitor design pattern in generating AST, and Interpreting it. For interpreting I made a Visitor Class, which is parent of Interpreter class. Visitor Class contains **virtual int visit function**, which is then defined in Interpreter Class. In BaseAst class parent contains **virtual int accept(Visitor* v**) function, which is then defined in all the classes used to generate AST.

5) Design of Interpreter

Structure of Vistor Class

```
class Visitor{
public:
  virtual int visit(class Prog* e) {}
  virtual int visit(class DeclList* e) {}
  virtual int visit(class Variables* e) {}
  virtual int visit(class Variable* e) {}
  virtual int visit(class StatementList* e) {}
  virtual int visit(class Terminal* e) {}
  virtual int visit(class Expr* e) {}
  virtual int visit(class ArithExpr* e) {}
  virtual int visit(class AssignExpr* e) {}
  virtual int visit(class BoolExpr* e) {}
  virtual int visit(class IfStmt* e) {}
  virtual int visit(class WhileStmt* e) {}
  virtual int visit(class GotoStmt* e) {}
  virtual int visit(class ForStmt* e) {}
  virtual int visit(class PrintStmt* e) {}
  virtual int visit(class ReadStmt* e) {}
```

Structure of Interpreter Class

```
class Interpreter:public Visitor{
public:
  Interpreter(){}
  int visit(class Prog* e);
  int visit(class DeclList* e);
  int visit(class Variables* e);
  int visit(class Variable* e);
  int visit(class StatementList* e);
  int visit(class Terminal* e);
  int visit(class Expr* e);
  int visit(class ArithExpr* e);
  int visit(class AssignExpr* e);
  int visit(class BoolExpr* e);
  int visit(class IfStmt* e);
  int visit(class WhileStmt* e);
  int visit(class GotoStmt* e);
  int visit(class ForStmt* e);
  int visit(class PrintStmt* e);
  int visit(class ReadStmt* e);
};
```

Each of the classes contained declared for AST contained **int accept(Visitor* v) { v->accept(this); }** , all the work for interpretation was defined in acc ept function.

6) Design of LLVM Code Generator

For genrating IR, I defined codegen() function in each classes defined for AST generation. And it contained the required LLVM code for genreating LLVM IR. Important thing is all the variables declared and used were globally declared.

7) Performance Comparison

A. Bubble Sort :- (for sorting 5 elements)

- **a. My Interpreter**
  ```
  real     0m0.015s
  user     0m0.008s
  sys      0m0.008s
  ```
- **b. lli**
  ```
  real    0m0.020s
  user     0m0.012s
  sys      0m0.004s
  ```
- **c. llc**
  ```
  real    0m0.065s
  user     0m0.016s
  sys      0m0.000s
  ```

B. Cumulative :- (defining and making cumulative of 10 numbers)

- **a. My Interpreter**
  ```
  real    0m0.012s
  user     0m0.000s
  sys      0m0.008s
  ```
- **b. lli**
  ```
  real    0m0.019s
  user     0m0.008s
  sys      0m0.008s
  ```
- **c. llc**
  ```
  real    0m0.020s
  user     0m0.012s
  sys      0m0.008s
  ```

C. Factorial :- (for calculating factorial of 5)

a. **My Interpreter**

```
real     0m0.011s
user    0m0.008s
sys     0m0.004s
```

b. **lli**

```
real    0m0.017s
user    0m0.012s
sys     0m0.004s
```

c. **llc**

```
real    0m0.018s
user    0m0.016s
sys     0m0.004s
```