

Compiler Design for FlatB language

Aman Mehta (201502106)

Project aim:

1. Writing parser for the new programming language with its given specification: Included writing scanner, writing grammar for the language and parsing it to detect syntactic errors.
2. A. Constructing abstract syntax tree for the grammar of the language, defining all the actions for the grammar and converting any code written in flatB to its corresponding AST using OOPS.
B. Building an interpreter to interpret and evaluate all the values for any code in flatB
3. Generating Intermediate Representation code for FlatB. Converting the AST to its IR so that this we can pass it to the backend and generate a binary executable using llc and interpreting our IR using lli.

Technologies used: flex, bison, llvm, llvm-as, lli, llc,

1. All the given specifications are implemented, except “codegen for read”. Everything given specification is implemented.

Integers and Array of Integers can be declared.

```
int data, array[100];  
int sum;  
Int x = 0;
```

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.

2. Expressions

a) for loop

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

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

```
for i = 2+5 (or any expression) ,100, 2 {
```

```
.....  
}
```

b) if-else statement

```
if bool-expression {  
.....  
}
```

```
if bool-expression {  
...  
}  
else {  
.....  
}
```

NOTE: It was asked to implement if <boolexpression> explicitly.

For eg: we cannot have,

```
If 1{  
  }  
}
```

Because 1 is not boolean. I have implemeted this feature too.

c) while statement

```
while bool-expression {  
  
}
```

d) conditional and unconditional goto

```
goto label  
goto label if bool-expression
```

e) print/read

```
print "blah...blah", val[*i+2], x  
println "new line at the end"  
read x
```

2. I have divided the main program into two obvious sections:

a. Declaration blocks: all declarations are handled by this

b. Codeblocks: all types of statements are handled here.

1) Declaration blocks contain many “declaration block” which include includes:

Int x,y; type

Int x=0,y=0; type

And int arr[100];

You can all the program variables by space separating them. (Refer to code grammar to actually visualize how I handled)

2) Codeblocks contain many “codeblock” and each codeblock may contain a state of type:

a. Assignment statement: a = <expr>; arr[<expr>] = <expr>;

b. If statements: if arr[<expr>] < expr2 or if <codeblocks> else <codeblocks> form.

c. While statement

d. Goto Statement

e. Print

f. Read

g. For statements. For i = <expr>, terminal_value, terminal_value(difference or jump)

So, for example, for a code: (I will explain on bubblesort.b during demo)

The grammar for this is as follows:

```
program: DECL '{' declblocks '}' CODE '{' codeblocks '}' {  
    $$ = new program($3,$7);  
    start = $$;  
}
```

```
terminal: ID { $$ = new unitClass("id", string($1)); }  
| NUMBER { $$ = new unitClass("num", $1); }
```

```
declblocks : { $$ = new declblocks(); }  
| declblocks declblock ';' { $$->push_back($2); }
```

```
declblock:  
    INT variables { $$ = new declblock($2); }
```

```
variables:  
    variable { $$ = new Vars(); $$->push_back($1); }  
| variables ',' variable { $$->push_back($3); }
```

```
variable:  
    ID { $$ = new Var(string("Normal"), string($1)); }  
| ID '[' NUMBER ']' { $$ = new Var(string("Array"),string($1),$3); }  
| ID '=' NUMBER { $$ = new Var(string("Normalnit"),string($1),"1",$3); }
```

```
| ID '=' '-' NUMBER { $$ = new Var(string("NormalInit2"),string($1),"1",-1*$4); }
```

```
codeblocks: { $$ = new codeblocks(); }  
| codeblocks codeblock { $$->push_back($2); }  
| codeblocks ID ':' codeblock { $$->push_back($4,$2); }
```

```
codeblock: Assign ';' { $$ = $1; }  
| If { $$ = $1; }  
| While { $$ = $1; }  
| Goto ';' { $$ = $1; }  
| Print ';' { $$ = $1; }  
| Read ';' { $$ = $1; }  
| For { $$ = $1; }
```

```
Last: ID { $$ = new last(string($1),string("Normal")); }  
| ID '[' exp ']' { $$ = new last(string($1),string("Array"),$3); }
```

```
exp : expr { $$ = new Expr("expr", $1); }  
| NUMBER { $$ = new Expr("num", $1); }  
| Last { $$ = new Expr("last", $1); }  
| '-' NUMBER { $$ = new Expr("num", -1*$2); }
```

```
expr: exp '+' exp { $$ = new binExpr($1,"+", $3); }  
| exp '-' exp { $$ = new binExpr($1,"-", $3); }  
| exp '*' exp { $$ = new binExpr($1,"*", $3); }  
| exp '/' exp { $$ = new binExpr($1,"/", $3); }
```

```
Assign: Last '=' exp { $$ = new Assign(($1), "=", $3); }  
| Last ADDEQ exp { $$ = new Assign(($1), "+=", $3); }  
| Last SUBEQ exp { $$ = new Assign(($1), "-=", $3); }
```

```
Type: EQEQ { $$ = new OperandType(string("==")); }  
| NOTEQ { $$ = new OperandType(string("!=")); }  
| MOREEQ { $$ = new OperandType(string(">=")); }  
| LESSEQ { $$ = new OperandType(string("<=")); }  
| '<' { $$ = new OperandType(string("<")); }  
| '>' { $$ = new OperandType(string(">")); }
```

```
BoolExp: exp Type exp { $$ = new boolExpr("expr", $1, $2, $3); }  
| BoolExp OR BoolExp { $$ = new boolExpr("bool", $1, "OR", $3); }  
| BoolExp AND BoolExp { $$ = new boolExpr("bool", $1, "AND", $3); }
```

```
If: IF BoolExp '{' codeblocks '}' { $$ = new ifStmt("if", $2,$4); }
    | IF BoolExp '{' codeblocks '}' ELSE '{' codeblocks '}' { $$ = new ifStmt("else", $2,$4,$8); }
```

```
While : WHILE BoolExp '{' codeblocks '}' { $$ = new whileStmt($2,$4); }
```

For :

```
FOR ID '=' exp ',' terminal '{' codeblocks '}' { $$ = new forStmt($2,$4,$6,$8); }
| FOR ID '=' exp ',' terminal ',' terminal '{' codeblocks '}' { $$ = new forStmt($2,$4,$6,$8,$10); }
```

```
Goto: GOTO ID IF BoolExp { $$ = new gotoStmt("cond", $2, $4); }
    | GOTO ID { $$ = new gotoStmt("uncond", $2); }
```

```
Read: READ Last { $$ = new readStmt($2); }
```

```
Print: PRINT Contents Content { $$ = $2; $$->type = 1; $$->push_back($3); }
    | PRINTLN Contents Content { $$ = $2; $$->type = 2; $$->push_back($3); }
```

```
Contents: { $$ = new printStmt(); }
    | Contents Content ',' { $$->push_back($2); }
```

```
Content : TOPRINT { $$ = new content($1,"string"); }
    | Last { $$ = new content($1,"last"); }
    | NUMBER { $$ = new content($1,"num"); }
```

3.

All the class and its method declarations are made in **classdefs.h** and all the functions are implemented in the main file **classes.cpp**

AST Hierarchy:

Base node is **astNode** and then it breaks as the grammar requirements and reaches leaf nodes. I have maintained a union node which says that all the objects created during parsing will be one of these class forms.

Owing to the size of the project, I have made modularised my code.

The hierarchy is as following:

```
class reportError (Handles all the error print statements throughout the codegen)
```

```
class astNode (root node)
```

class OperandType: public astNode

class unitClass:public astNode

class program:public astNode

class declblocks:public astNode

class declblock:public astNode

class Vars:public astNode

class Var:public astNode

class codeblocks:public astNode

class codeblock:public astNode

class boolExpr:public astNode

class Expr:public astNode

class ifStmt:public codeblock

class whileStmt:public codeblock

class gotoStmt:public codeblock

class forStmt:public codeblock

class readStmt:public codeblock

class printStmt:public codeblock

class content: public codeblock

class Assign:public codeblock

class last: public Expr

class binExpr:public Expr

4. I have used interpreter class to implement interpreter for all the classes in a single class. I am accessing this function using accept function, common in all the classes whichever needs to be visited. If I had not used visitor design, I would have had to implement visit functions for all the classes separately in their own class. Interpreter class gives me the design to handle all the visit functions together (which are mostly similar and hence working on them in a single class helps)

5. Interpreter runs parallel with AST. It first sees the type of node present on stack and calls its accept function to visit it. After completion, the accept function pushes the required values on stack and now the stack again repeats the same process, until it is not empty.

The class which defines all the visit functions for all the classes : **class Visitor**

The class which implements all the visit functions for all classes : **class Interpreter:public Visitor**

object->accept() function is called whenever that object is required to be visited.

6. I have implemented LLVM code generators for each class. My code can be considered as a big black box, which is a object of “program” class and has only 2 children “declblocks” and “codeblocks”. Now, you go to this node and further call for code generation of declblocks. This further opens up the “declblocks” black-box into its children by calling their code generators. This IR generation is sort of a DFS, and once you reach to a node, i.e. a point where you can write the required portion in MIPS or x86 instructions, you add respected load, store, add, branch, etc operations and return back to the parent to take care of other children.

7. I have the outputs saved on my machine for bubble sort, factorial and cumulative sum codes.

Performance:

A. **Factorial:** (USING GOTO)

a. LLC

- i. real 0m0.021s
- ii. user 0m0.012s
- iii. sys 0m0.008s

b. LLI

- i. real 0m0.020s
- ii. user 0m0.012s
- iii. sys 0m0.004s

c. My interpreter

- i. real 0m0.008s
- ii. user 0m0.004s
- iii. sys 0m0.004s

B. Bubblesort: (WHILE, FOR, ARRAY comparisons)

a. LLC

- i. real 0m0.026s
- ii. user 0m0.020s
- iii. sys 0m0.004s

b. LLI

- i. real 0m0.025s
- ii. user 0m0.020s
- iii. sys 0m0.004s

c. My interpreter:

- i. real 0m0.018s
- ii. user 0m0.008s
- iii. sys 0m0.008s

C. Cumulative sum (FOR, WHILE, ARRAYS and operations)

a. LLC

- i. real 0m0.013s
- ii. user 0m0.000s
- iii. sys 0m0.016s

b. LLI

- i. real 0m0.011s
- ii. user 0m0.008s
- iii. sys 0m0.000s

c. My interpreter

- i. real 0m0.020s
- ii. user 0m0.012s
- iii. sys 0m0.004s

This project gave me a deep understanding of how any code in a specific programming language is executed and interpreted and looked for errors by its compiler.

After completing this project and looking at the outputs of bubblesort and etc in this new programming language FlatB gives a great feeling of satisfaction.