Flat B Compiler Report

Name: *Pemmaraju Sai Vasishth*
Roll Number: *201501179*

Programming Language Description

All the statements are ended by a semicolon.
It has two parts. Declaration block and Code Block. They standard template of the code will be:

```
declblock
{
        <Declarations>
}
codeblock
{
        <Code>
}
```

We have to declare all our variables in declblock only. All the declarations are to be in int. There are 2 types of variables. Normal variable and array.Arrays are to be declared with a fixed numeric size. Variables declared in declblock are used in codeblock. Only positive integers are taken as input.

Eg.

```
int a;
int array[100];
int a,b,c,temp[200];
```

In the codeblock, we use expressions, string(only for printing), bools, label names. Here, expression is an arithmetic computation between two other expressions(whatall are allowed is put written in CFG description). Bools is the comparison of 2 expressions. Label names are just names to name a particular line in the code.

In the codeblock, there are the following statements:

1. Assignment

    variable = <expression>; (or) variable[<expression>] = <expression>;

The value computed in rhs is stored in the lhs.

2. Printing

print <string>, <expression>,....;

Printing ends with newline by default.

Consecutive Values are printed using comma as separation.

3. Reading

read variable1,variable2,.....,variablen[<expression>],....;

Multiple inputs can be read.

Values are read in a variable or at an array location.

4. For loop

for var = <expression1>,<expression2> {<Code>}

OR

for var = <expression1>,<expression2>,<expression3> {<Code>}

Expression1 here evaluates to the start point of the for loop.

Expression2 evaluates to the end point of the for loop.(goes from start to end)

Expression evaluates to the step or increment of the loop.

Here var is just a dummy name. It can be any name.

It is not used anywhere in the code and doesn't matter what it is.

5.While loop

while <bool> { <Code> }

This loop keeps running until the boolean statement is false.

6. If-else

if <bool> { <Code> }

OR

if <bool> { <Code> } else { <Code> }

7. Labeling

<label-name> : <Code>

This is just to name the immediate statement in the code.

8. Goto Unconditional

got <label-name> ;

Unconditional jump to the statement that is represented by this label-name.

9. Goto Conditional

goto <label-name> if <bool> ;

Jump to the statement that is represented by this label-name only if bool is true.

Syntax Analysis

Syntax analysis is done for checking the syntax of the code. Bison is used to specify the Context free grammar and automatically generate the parser. Lexer is used to generate tokens for specific patterns, it acts like a regex. Parser along with lexer will check the syntax of the grammar.

Bison uses LALR(1) grammar to generate the Automaton and parsing

table with will be used to check the syntax.

Examples:

Program: declblock Decls_B codeblock Code_B

This is the start of the code. Decls_B represents the contents of the declaration block and Code_B represents the code block.

```
Decls_B:  { Decls }
Decls:  Decls Decl SC |
Decl: INT Vars
Vars:   Vars COMMA Var | Var
Var:   IDENTIFIER | IDENTIFIER OSB NUMBER OCB
```

This is the grammar for the declaration block. Here, INT, IDENTIFIER, NUMBER, SC, COMMA, OSB, OCB are defined in the lexer.
There will be a syntax error if the grammar is not followed or the token is not defined in the lexer.
Similarly, this is the context free grammar for the code block.

```
Code_B:   OB Stmnts CB
Stmnts: Stmnts Stmnt | ;
Stmnt:  Assign SC
        | Printing SC
        | Reading SC
        | Forloop
        | Whileloop
        | Ifelse
        | Got1 SC
        | Got2 SC
        | Labeling

Bool   :  OP Bool CP
       |  Expr E Expr
       |  Expr NE Expr
       |  Expr GT Expr
       |  Expr LT Expr
       |  Expr GE Expr
       |  Expr LE Expr
```

```
Expr   :       OP Expr CP
       |       Expr ADD Expr
       |       Expr MUL Expr
       |       Expr SUB Expr
       |       Expr DIV Expr
       |       Expr MOD Expr
       |       Expr XOR Expr
       |       Expr AND Expr
       |       Expr OR Expr
       |       NUMBER
       |       IDENTIFIER
       |       IDENTIFIER OSB Expr OCB
       ;
```

Design of AST

During Parsing, we construct abstract syntax tree for our code. We Use classes to construct nodes for each type. As Bison is following LALR(1) grammar ie., a bottom-up parser, so the classes get constructed from bottom to top ie., from terminal to its parent non-terminal and so on. Here, all classes have astNode as their parent class.
The classes used in the code are:

- class astNode { };
- class Prog:public astNode { };

For declaration block:

- class Decls_block:public astNode { };
- class Decls:public astNode { };
- class Decl:public astNode { };
- class Vars:public astNode { };
- class Var:public astNode { };

For code block:

- class Expr:public astNode { };
- class Assign:public astNode { };
- class Ident:public astNode { };
- class Var1:public astNode { };
- class Printing:public astNode { };

- class Identr:public astNode { };
- class Varr:public astNode { };
- class Reading:public astNode { };
- class Var2:public astNode { };
- class Forloop:public astNode { };
- class Bool:public astNode { };
- class Whileloop:public astNode { };
- class Ifelse:public astNode { };
- class Labeling:public astNode { };
- class Got1:public astNode { };
- class Got2:public astNode { };
- class Statement:public astNode { };
- class Statements:public astNode { };
- class Code_block:public astNode { };

Semantic Analysis

After construction of the AST, we traverse through the tree and check for semantics like: Redeclaration of a variable, checking whether it is declared or not, same goes for the labels, check if the array is out of bounds.

Visitor Design Pattern

Visitor Design Pattern is a way of separating an algorithm from an object structure from which it operates. Visitor allows adding new virtual functions to a family of classes,without modifying the classes.It takes object reference as input and implements the virtual function.
In our code, a visitor class is created which contains all the virtual functions to all the family of classes where each function takes object reference of that class as input. Next, we can use any class as the child class of visitor class to perform actions on already defined functions without altering them. For example, the interpreter class has visitor as its parent class.

Design of Interpreter

Interpreter is implemented using visitor design pattern. In our code, we traverse through the code by calling visit function accept() in each class. We create a symbol_table using map(c++ STL), and store all our variables and labels in it. Using this, we can interpret a

code and also do error checking by checking for variables and labels from our symbol_table.

Example:

```
void interpreter::visit(class Forloop* forloop)
{
        int start=0,end=-1,step=1;
        start = forloop->start->accept(v);
        end = forloop->end->accept(v);
        if(forloop->step != NULL) step = forloop->step->accept(v);
        for(;start<=end;start+=step) forloop->stmnts->accept(v);
}
```

Interpreter Code for for loop.

Design of LLVM Code Generator

An LLVM module class instance acts as a container which can store all other LLVM IR objects. When we create instructions using the LLVM Builder class, the method gets dumped in module instance. Finally when we call Module's instance dump method, we get entire IR code as output. A Codegen() method is added to every class and we use the code generation API in them to generate our code accordingly.

Creation of LLVM Module Class:

```
static Module *ModuleOb = new Module("Bcomp",llvm::getGlobalContext());
```

IRBuilder class provides an API to create instructions.

```
static LLVMContext &Context = getGlobalContext();
static IRBuilder<> Builder(Context);
```

Example:

CODE:

```
declblock
{
        int i;
}
codeblock
```

```
{
	i = 4;
	for var = 5,9
	{
		if (i < 8) { print "BAD"; }
		i = i+1;
	}
}
```

Generated Code:

```
; ModuleID = 'Bcomp'

@i = common global i32 0, align 4
@_iterator1 = common global i32 0, align 4
@0 = private unnamed_addr constant [8 x i8] c"\22BAD\22 \0A\00"

define void @main() {
entry:
  store i32 4, i32* @i
  store i32 5, i32* @_iterator1
  %0 = load i32, i32* @_iterator1
  %limit = icmp ule i32 %0, 9
  br i1 %limit, label %loop, label %afterloop

loop:                                             ; preds = %ifcont, %entry
  %1 = load i32, i32* @i
  %ltcomparetmp = icmp ult i32 %1, 8
  br i1 %ltcomparetmp, label %if, label %else

afterloop:                                        ; preds = %ifcont, %entry
  ret void

if:                                               ; preds = %loop
  %2 = call i32 @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]* @0, i32 0, i32 0))
  br label %ifcont

else:                                             ; preds = %loop
  br label %ifcont

ifcont:                                           ; preds = %else, %if
```

```
  %3 = load i32, i32* @i
  %addtmp = add i32 %3, 1
  store i32 %addtmp, i32* @i
  %4 = load i32, i32* @_iterator1
  %nextval = add i32 %4, 1
  store volatile i32 %nextval, i32* @_iterator1
  %limit1 = icmp ule i32 %nextval, 9
  br i1 %limit1, label %loop, label %afterloop
}

declare i32 @printf(i8*)
```

After the IR Code generator, we generate the llvm bit code by using llvm-assembler. lli is used to interpret the code generated. llc is used to convert the llvm bit code to target assembly code.

Performance Comparison

1. Bubble Sort. [array size 10 initialised with random integers. 100000 test cases]
   A. My interpreter
      I. Number of instructions: 49,82,50,05,318
      II. Wall Clock Time: 12.634114858 seconds.
   B. LLI
      I. Number of instructions: 4,72,64,26,662
      II. Wall Clock Time: 3.916405521 seconds.
   C. LLC
      I. Number of instructions: 4,59,85,24,464
      II. Wall Clock Time: 3.560332044 seconds.

2. GCD of two numbers. [random integers given. 100000 test cases]
   A. My interpreter
      I. Number of instructions: 6,17,78,48,469
      Ii. Wall Clock Time: 1.659853168 seconds
   B. LLI
      I. Number of instructions: 56,55,58,694
      Ii. Wall Clock Time: 0.355008585 seconds

C. LLC
    I. Number of instructions: 53,13,55,393
    Ii. Wall Clock Time: 0.330439883 seconds

3. A^B mod N [random integers given. 1000000 test cases]
    A. My interpreter
        I. Number of instructions: 78,02,93,75,633
        Ii. Wall Clock Time: 25.138860705 seconds
    B. LLI
        I. Number of instructions: 5,83,95,31,503
        Ii. Wall Clock Time: 4.528283207 seconds
    C. LLC
        I. Number of instructions: 5,71,55,22,141
        Ii. Wall Clock Time: 3.703890479 seconds

Using LLC and generating the executable and executing it will take less time when compared to using LLI, which is in turn faster than my own interpreter. However, when running very small and trivial codes like printing hello world or running a small loop of 10 numbers, my interpreter interprets the code faster than lli. However, using llc executable is always fastest among these three.