Compiler-Design

Description of FlatB Language

FlatB is a general purpose programming language, supporting structured programming.

- Only unsigned Integer and array of unsigned integer data-types are allowed.
    - Eg: int data,a[100];
  - All the variables should be declared in the declblock{.....} before being used in the codeblock{....}. Multiple variables can be declared in each statement in declblock{}
  - Each statement ends with a semicolon at the end.
- All Arithmetic and logical operations are allowed.
- For loop syntax:
  - for i = 1,100 {

    }
    Here, 1 is start value,100 is end value;cond is i<=100
  - for i = 1,100,2{

    }
    Here,2 is increment value;

Variables and expressions can also be used in place of numbers.

- While Loop syntax:
  - while expression{

    }
- If-else syntax:
  - If expression {

    }

    - If expression{

      }
      else{

      }
- Conditional and Unconditional goto:
    - goto label;
    - goto label if expression;

  Here, label can be defined anywhere in the codeblock, label is used to indicate start of specific statement.
- Print
    - print “any string”,var1,var2,....
    - println “any string”

  If u use println, it outputs a newline at end, print doesn’t give newline.
- Read
    - read var1,var2,.....

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.

Example:
1)

- *program: decl_block code_block*
    - This is start of the code, it says 2 blocks decl_block and code_block
- decl_block: declblock '{' declaration_list '}'
    - Here, this states CFG for decl_block should be declblock{...}

- declaration_list : decl_stmt ';' declaration_list
  |
  ;

- decl_stmt : TYPE expr
  ;

- expr : variables ',' expr
  | variables
  ;

- variables : IDENTIFIER '[' VALUE ']'
  | IDENTIFIER
  ;
- VALUE : DIGIT
  | NUMBER
  ;

The above grammar is for declaration block , Here, IDENTIFIER, TYPE, DIGIT, NUMBER are tokens defined in lexer(scanner.l) and if the input statements in declaration block doesn't satisfy this syntax, then parser generates a syntax error.

2)

- statement_list : forloop statement_list
  | whileloop statement_list
  | gotoloop ';' statement_list
  | condition statement_list
  | IDENTIFIER ':' statement_list
  | print ';' statement_list
  | read ';' statement_list
  | stmt ';' statement_list
  |
  ;

This is the grammar for different statement list types in codeblock.

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.

Class hierarchy for my AST design is as follows:

- class Astnode{...};
- class Program:public Astnode{...};

For Declaration block

- Class Decl_block:public Astnode{...};
- class Declaration_list:public Astnode{...};
- class Decl_stmt:public Astnode{...};
- class Variables:public Astnode{...};

For Code block

- class Code_block:public Astnode{...};
- class Expr2:public Astnode{...};
- class Variables2:public Astnode{...};
- class Expression:public Expr2{...};
- class BoolExpression:public Expr2{...};
- class Statements:public Astnode{...};
- class Stmt:public Statements{...};
- class Statement_list:public Astnode{...};
- class Condition:public Statements{...};
- class WhileLoop:public Statements{...};
- class ForLoop:public Statements{...};
- class Print:public Statements{...};
- class Read:public Statements{...};
- class GoToLoop:public Statements{...};

Semantic Analysis

After constructing Abstract syntax tree, we traverse through it and check for semantics:

1) Check whether the variable is declared or not.
2) Check whether the label is declared or not.
3) Check for the redeclaration of label.
4) Check for redeclaration of variable.
5) Check whether the array access is out of bound or not.

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.

Implementation:

- Visitor class is created which contains the virtual functions to all the family of classes where each function takes object reference of that class as input.
- After that interpreter class is created which has visitor class as parent class.
- This is quite useful in cases where we don't need to alter the already defined functions, but perform operations on it.
- Any other type of operations can be implemented by creating a class for those functions which perform that type of operations on those family of classes. This class has visitor class as parent class.

Design of Interpreter

Interpreter is implemented by using visitor design pattern.

Implementation:

- In interpreter, we traverse through the code by calling visit functions in each class.
- We create a class named SymbolTable which contains a map instance as variable, which contains all the variable names mapped with a struct node defined for storing all the variable data.
- SymbolTable class also contains functions that can be used to modify the value of a particular variable. It also helps in semantic checking.
- We also create a class named SymbolTable2 which contains a map instance as variable, which links all the label names with statementlist class pointer which is the start of that particular label.
- It also contains functions for checking whether the label is declared or not.

Example:

```
int Interpreter::visit(class WhileLoop* vis_var)
{
  int temp_check;
  vis_var->val = vis_var->reqd_expr->accept(Vis);
  while(vis_var->val==1)
  {
        temp_check = vis_var->list1->accept(Vis);
        if(temp_check==-1)
        return -1;
        vis_var->val = vis_var->reqd_expr->accept(Vis);
  }
  return 0;
}
```

The above is the interpreter implementation for WhileLoop class.

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.
Constructor for LLVM Module Class:

```
static Module *TheModule = new Module("FlatB Compiler",Context);
```

We also have IRBuilder class which can provide the API to create instructions for all the methods used.
Constructor for LLVM IRBuilder class:

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

Example:

```
Value* WhileLoop::codegen()
{
  Value *V = ConstantInt::get(getGlobalContext(),APInt(32,0));
  Value *req_val = reqd_expr->codegen();
  Function *req_func = Builder.GetInsertBlock()->getParent();
  BasicBlock *loopBB = CreateBB(req_func,"loopBlock");
  BasicBlock *nextBB = CreateBB(req_func,"nextBlock");
  Builder.CreateCondBr(req_val,loopBB,nextBB);
  Builder.SetInsertPoint(loopBB);
  Value *loop_val = list1->codegen();
  if(loop_val==0)
        return 0;
  req_val = reqd_expr->codegen();
  Builder.CreateCondBr(req_val,loopBB,nextBB);
  Builder.SetInsertPoint(nextBB);
  return V;
}
```

If Code is as follows:

```
declblock{
        int data[100];
        int i,sum;
}
codeblock{
        i = 10;
        sum = 0;
        if i==10 {
                while i<13 {
                i = i + 1;
                sum = sum + i;
                }
        }

        print "i: ", i , "Sum: ",sum;
}
```

IR code generated is as follows:

```
; ModuleID = 'FlatB Compiler'

@sum = common global i32 0, align 4
@i = common global i32 0, align 4
@data = common global [100 x i32] zeroinitializer, align 4
@0 = private unnamed_addr constant [21 x i8] c"i: \22, i , \22Sum:  %d \00"
@1 = private unnamed_addr constant [17 x i8] c"i: \22, i , \22Sum: \00"

define i32 @main() {
entry:
  store i32 10, i32* @i
  store i32 0, i32* @sum
  %0 = load i32, i32* @i
  %CE = icmp eq i32 %0, 10
  %ifCond = icmp ne i1 %CE, false
  br i1 %ifCond, label %ifBlock, label %elseBlock

ifBlock:                                          ; preds = %entry
  %1 = load i32, i32* @i
```

```
  %CLT = icmp ult i32 %1, 13
  br i1 %CLT, label %loopBlock, label %nextBlock

elseBlock:                                        ; preds = %entry
  br label %contBlock

contBlock:                                        ; preds = %elseBlock, %nextBlock
  %2 = load i32, i32* @sum
  %3 = call i32 @printf(i8* getelementptr inbounds ([21 x i8], [21 x i8]* @0, i32 0,
i32 0), i32 %2)
  ret i32 0

loopBlock:                                        ; preds = %loopBlock, %ifBlock
  %4 = load i32, i32* @i
  %AddOp = add i32 %4, 1
  store i32 %AddOp, i32* @i
  %5 = load i32, i32* @sum
  %6 = load i32, i32* @i
  %AddOp1 = add i32 %5, %6
  store i32 %AddOp1, i32* @sum
  %7 = load i32, i32* @i
  %CLT2 = icmp ult i32 %7, 13
  br i1 %CLT2, label %loopBlock, label %nextBlock

nextBlock:                                        ; preds = %loopBlock, %ifBlock
  br label %contBlock
}

declare i32 @printf(i8*, i32)
```

After the IR code generation,

- We can generate the llvm bit code by using llvm-assembler.
- After that we can use lli to interpret the code generated.
- We can use llc to convert llvm bit code to target assembly code, and if that target architecture is not specified, it uses architecture of host system.

Performance Comparison

- For Bubble Sort : {For n = 1000}
    - Using My Interpreter:
        - Number of Instructions: 3,49,39,63,340
        - Wall Clock Time: 0.452563948 seconds
    - Using lli:
        - Number of Instructions: 3,69,18,873
        - Wall Clock Time: 0.062287797 seconds
    - Using llc:
        - Number of Instructions: 90,48,603
        - Wall Clock Time: 0.008078862 seconds
- For Printing Pyramid: {For n = 1000}
    - Using My Interpreter:
        - Number of Instructions: 4,87,04,14,960
        - Wall Clock Time: 1.038770072 seconds
    - Using lli:
        - Number of Instructions: 55,43,16,080
        - Wall Clock Time: 0.378646243 seconds
    - Using llc:
        - Number of Instructions: 46,95,42,682
        - Wall Clock Time: 0.353385636 seconds
- For nCr: {for n = 10 and r = 5 and t = 100000 }
    - Using My Interpreter:
        - Number of Instructions: 6,38,13,64,483
        - Wall Clock Time: 1.046717401 seconds

- Using lli:
  - Number of Instructions: 48,28,47,595
  - Wall Clock Time: 0.240374409 seconds

- Using llc:
  - Number of Instructions: 44,66,70,433
  - Wall Clock Time: 0.221931102 seconds

Here, we observe that parsing time is around 0.007595398 seconds and number of instructions required is around 41,07,423.

We also observe that using llc and generating the code in host architecture and then executing it will take less time when compared to that executed using lli which is far better when compared to the interpreter made using Abstract syntax tree.