Compiler Design

Description of FlatB Language

FlatB is a general purpose programming language,supporting the programme structure as follows :

- Supports only unsigned integers and integer arrays
- Arithmetic and logical operations are allowed
- If loop syntax:
    - If boolean expression {
      ---statement block------
      }
      else {
      ---statement block------
      }

    - If boolean expression {
      -----statement block-----
      }
- For loop syntax:
    - for i=0, n {
      ---statement list----
      }
    - For i=0, n, 2{
      ---statement list----
      }
- While Loop syntax:
    - while expression{

      }

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

- 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 code.Bison is used to specify context free grammar and automatically generate parser.
Lexer generates grammar tokens which are used in bison file and basically acts like a regex.Lexer combined with parser checks semantic analysis.

Example:

- *Program : Decl_block Code_block*
    - This is where the code starts, It defines program contains two blocks declaration block and code block

- *Decl_block : declblock ‘{‘ decl_list ‘}’*
    - The above grammar describes the structure of Decl_block as the string declblock and decl_list variables along with parenthesis

- *decl_list : decl_list decl_data | decl_data*

    - This specifies the structure of decl_list as vectors of decl_data

- *decl_data : INT variables ‘;’*
    - Defines the structure of decl_data

- variables : IDENTIFIER | IDENTIFIER ‘[‘ number ‘]’
    - Defines the variable grammar

IDENTIFIER,number,INT are defined in lexer and passed as tokens form ,if Input programme doesn’t follow the CFG then parser generates error. In the same way Code_block is defined with statement_list.

- *Statement_list : Statement_list Forms*
  | Forms ;
- *Forms : While*
  | For
  | Assign
  | If
  | Goto
  | Label
  | COUT Printparts ';';

- *Printparts : Printparts ',' Printpart*
  | Printpart

- *Printpart : Vars*
  | SENTENCE
  | NUMBER ;

- *Vars : IDENTIFIER ;*
  | IDENTIFIER '[' Expr ']' ;

Similarly the grammar is written in parser.y

The above stated grammar is self explained

Example of Code parsing through grammar:

- Declblock
  {
  int a, b;
  }
  Codeblock
  {
  a=b;
  }

- In the above stated example initially the parser passes the code into programme grammar which has two blocks declblock ad codeblock,Then the statement ‘int a,b’ is passed through decl_list,decl_data and finally INT IDENTIFIER which accepts the statement .
- Similarly when the code is parsed through code block it is passed through st_list,Forms and finally Assign grammar which accepts it.

- In the similar way the code with correct syntax is accepted otherwise rejected.

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 Ast is as following:

- Class Astnode {.....}
    - Class Prog : public Astnode {.....}

    - Class Decl_list : public Astnode {.....}
    - Class Decl_data : public Astnode {.....}
    - Class Var_s : public Astnode {.....}
    - Class Var : public Astnode {....}

    - Class St_list : public Astnode {....}
    - Class Forms : public Astnode {.....}
        - Class Assign : public Forms {.....}
        - Class For_l : public Forms {.....}
        - Class While_l : public Forms {.....}
        - Class Ifelse : public Forms {.....}
        - Class recPrint : public Forms {.....}
        - Class Printpart : public Forms {.....}
            - Class SenPrint : public Printpart {.....}
            - Class VarPrint : public Printpart {.....}
            - Class NumPrint : public Printpart {.....}

        - Class Goto : public Forms {.....}
        - Class Label : public Forms {.....}

    - Class Expr : public Astnode {.....}
        - Class binExpr : public Expr {.....}
        - Class unExpr : public Expr {.....}

- Class enclExpr : public Expr {.....}

For boolean and condition it is declared similar to Expr

Semantic Analysis

After constructing Ast, we traverse through it and check for semantics :

- Check whether the variable is declared or not.
- Check whether the variable is declared or not.
- Check for the redeclaration of label.
- Check for redeclaration of variable.

Design of Semantic Analysis :

A class Table is created inorder to check for semantics, the table class is written as Follow:

- Class Table {
  Public:
  ----
  def insert(.....) :
  ----
  def lookup (.....):

  }

- Here the insert functions constructs a variable list vector and stores them, basically done only in declblock
- The lookup function checks whether the variable used in codeblock is whether present in the declblock or not , also checks the redeclaration of variables in declblock.

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 similarly in the way of AST construction.
- In interpreter, we traverse through the code by calling visit functions in each class.
- We created a class named VarTable , which contains all the variable names mapped with a struct node defined for storing all the variable data.
- We use the update() function for VarTable to update values during the interpretation.

Design Of llvm codegen

An llvm module class instance acts as container and stores the llvm IR object.When we create instructions using llvm Builder class, methods get dumped in module instance.Finally when we call Module's instance dump method, we get entire IR code as output. Constructor for llvm's module class:

static Module *The Module = new Module ("B compiler", Context)

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

Constructor for llvm IRBuilder class :

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

The codegen basically generates assembly language.

Example:
For example consider assign statement

- a=9;
  Now “ store i32 9, i32* @a “ is the instruction for the above line,
  In llvm we write
  Builder.CreateStore(9,a);
- a=b;
  Now “ %0 = load i32, i32* @j is instruction for assigning a=b,
  store i32 %0, i32* @i ”
  In llvm we write
  Value* b= Builder.CreateLoad(b);
  Builder.CreateStore(b,a);

The output of codegen contains assembly code in form of Basicblocks, LLVM API provides us the library which creates basic blocks
*llvm::Function *mainFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", TheModule);*

Here we create mainFunc as object of Function class which stores basic blocks in sequence they are inserted

*llvm::BasicBlock *entry = llvm::BasicBlock::Create(Context, "entrypoint", mainFunc);*

This creates the First basicblock which intializes the codeblock of our language

*BasicBlock *ifBlock = BasicBlock::Create(getGlobalContext(), "if", mainFunc);*

This is the basic syntax for creating the Basicblock of ifBlock, we write similarly for elseblock, while and goto blocks

We use the above mentioned BasicBlock functions, Loading and storing and some other basic API functions to create codegen for our language.

Performance Comparison

- For Bubble Sort : {For n = 1000}
  - Using lli:
    - Number of Instructions: 3,82,70,826
    - Wall Clock Time: 0.015325590 seconds
  - Using llc:
    - Number of Instructions: 1,18,16,786
    - Wall Clock Time: 0.007444635 seconds

- For Seives Method to compute Primes (n=1000) :
  - Using lli:
    - Number of Instructions: 2,56,68,707
    - Wall Clock Time: 0.013433946 seconds
  - Using llc:
    - Number of Instructions: 6,80,608
    - Wall Clock Time: 0.001472190 seconds

- For GCD : {For two random numbers }
  - Using lli:
    - Number of Instructions: 2,28,35,146
    - Wall Clock Time: 0.014852702 seconds
  - Using llc:
    - Number of Instructions: 5,19,906
    - Wall Clock Time: 0.001521339 seconds

The **llc** command compiles LLVM source inputs into assembly language for a specified architecture. The assembly language output can then be passed through a native assembler and linker to generate a native executable.

So therefore we observe that llc performance is better than lli which is not native to system