Compilers Project Report

By : Vinaya Khandelwal (201401139)

FlatB Programming Language Description :-

The language consists of two main blocks. The declare block contains all the code required to declare the integer variables and arrays used in the program. The code block holds the code necessary to perform operations on the declared variables and achieve the required output. The following functionality is present in my flatB compiler.

1. Data Types - Integers and Array of Integers.

2. Arithmetic Expressions - Add , Sub , Mul
Logical - And , Or
Relational - (>,<.==,!=,>=,<=)

3. For loop - 2 variants

4. If and If-Else statement

5. while loops

7. print/read

```
print "blah...blah", val
println "new line at the end"
read sum
read data[i]
```

Syntax and Semantics :-

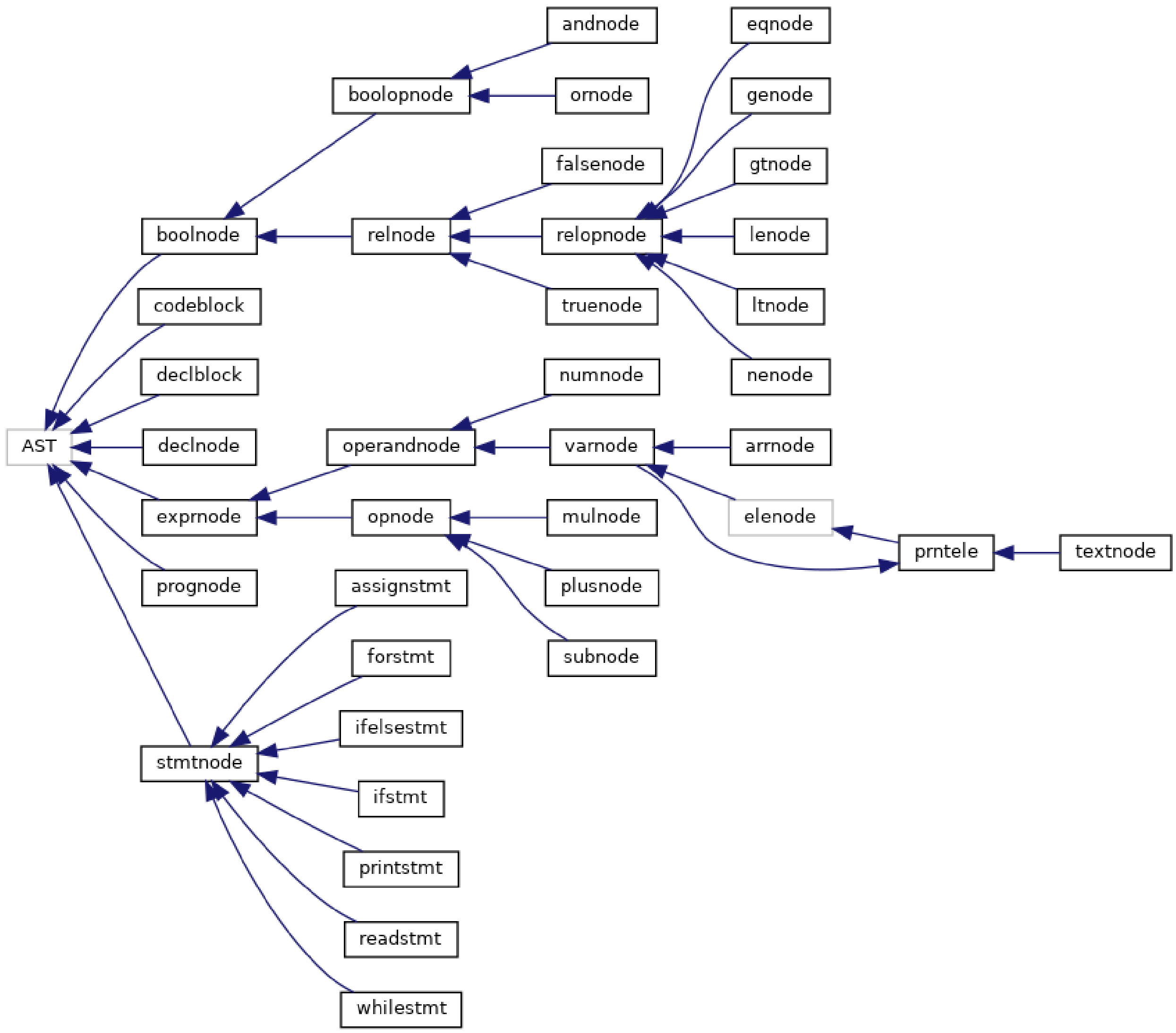
```
program:        DB decl_block CB code_block

decl_block:      '{'declaration_list'}'
          |       '{'  '}'             ;

declaration_list:     declaration
```

```
        |       declaration_list declaration
                ;

declaration:    DATATYPE instance ';'
                ;

instance:       instance ',' variable
        |       variable                ;

code_block:     '{'statement_list '}'
        |       '{' '}'
                ;

statement_list:  statement
        |       statement_list statement
        |       statement_list LABEL ':' statement


statement:      IF boolexpr code_block
        |       IF boolexpr code_block ELSE code_block
        |       WHILE boolexpr code_block
        |       FOR IDENTIFIER '=' operand ',' operand code_block
        |       FOR IDENTIFIER '=' operand ',' operand ',' operand code_block
        |       GOTO LABEL IF boolexpr ';'
        |       GOT LABEL ';'
        |       variable '=' expr ';'
        |       PRINT element ';'
        |       PRINTLN element ';'
        |       READ variable ';'
                ;

expr:           expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       operand
                ;

boolexpr:       boolexpr AND boolexpr
        |       boolexpr OR boolexpr
```

```
          |     relexpr
          ;

relexpr:        operand GT operand
          |     operand LT operand
          |     operand GE operand
          |     operand LE operand
          |     operand EQ operand
          |     operand NE operand
          |     TRUE
          |     FALSE
                ;

element:        element ',' printables
          |     printables
          ;

printables:     variable
          |     TEXT
          ;

operand:        NUMBER
          |     variable
          ;

variable:       IDENTIFIER
          |     array
          ;

array:          IDENTIFIER '[' NUMBER ']'
          |     IDENTIFIER '[' IDENTIFIER ']'
          ;
```

Design of AST :-

Generated Using Doxygen Wizard. More information available at file:///home/mayukuse/html/index.html


andnode
eqnode
boolopnode
ornode
genode
falsenode
gtnode
boolnode
relnode
relopnode
lenode
codeblock
truenode
ltnode
declblock
numnode
nenode
AST
declnode
operandnode
varnode
arrnode
exprnode
opnode
mulnode
elenode
prntele
textnode
prognode
assignstmt
plusnode
forstmt
subnode
ifelsestmt
stmtnode
ifstmt
printstmt
readstmt
whilestmt

Visitor Design Pattern :-

The visitor design pattern is very useful for the construction of the compiler. The following benefits are obtained from the usage of the pattern :-

1) The memory consumed by the AST is reduced. Only required nodes in the AST are created which contain the necessary functionality.
2) It allows us to separate the data structure from the algorithms. An operation can be applied to all the objects in the AST without modifying the AST.
3) Also, from point 2, we can also add multiple passes to the AST to perform different operations by only adding the extra functionality. Prevents the creation of redundant code.

Interpreter Design :-

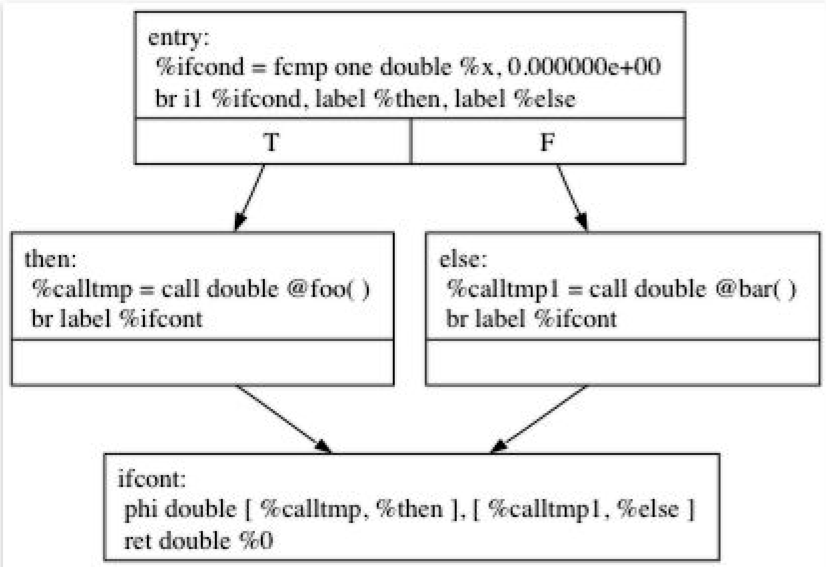
The interpreter evaluates the data being held in the AST nodes. It uses C++ data structures and operations for the purpose of evaluating the nodes.

Code Generation Design :-

The code generation is performed using the LLVM compiler infrastructure. LLVM provides tools for the purpose of compiler construction. The LLVM code generation tool was used for this project to generate the IR representation of the Bcc input program.

Major components of the LLVM IR generation involves the Module, Function, BasicBlocks and Instructions. Libraries like the Context have been used to provide the necessary LLVM data types and constant values. The IRBuilder has been used to ease the creation of instructions and to connect the different components mentioned above.

Example of Control Flow graph required in generation of If-Else and Loops.



CFG for 'baz' function

Performance Comparison:-

1.) Bubble sort on first 10 integers originally in descending order
Performance counter stats for './bcc ../test-units/llvmtest.b':

24.673016	task-clock:u (msec)	#	0.986 CPUs utilized
0	context-switches:u	#	0.000 K/sec
0	cpu-migrations:u	#	0.000 K/sec
1,452	page-faults:u	#	0.059 M/sec
35,262,891	cycles:u	#	1.429 GHz
63,027,376	instructions:u	#	1.79 insn per cycle
8,723,081	branches:u	#	353.547 M/sec
171,596	branch-misses:u	#	1.97% of all branches

0.025021542 seconds time elapsed

2) Nested For loop of depth 3
Performance counter stats for './bcc ../test-units/triplefor.b':

```
        113.361834      task-clock:u (msec)       #    0.995 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             1,454      page-faults:u             #    0.013 M/sec
       292,893,951      cycles:u                  #    2.584 GHz
       768,967,651      instructions:u            #    2.63  insn per cycle
       204,730,123      branches:u                # 1805.988 M/sec
           187,838      branch-misses:u           #    0.09% of all branches

       0.113921472 seconds time elapsed
```

lli) Nested For loop of depth 3
Performance counter stats for 'lli output.ll':

```
         28.164955      task-clock:u (msec)       #    0.987 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             1,850      page-faults:u             #    0.066 M/sec
        38,808,283      cycles:u                  #    1.378 GHz
        53,767,013      instructions:u            #    1.39  insn per cycle
         9,491,346      branches:u                #  336.991 M/sec
           260,275      branch-misses:u           #    2.74% of all branches

       0.028528978 seconds time elapsed
```

llc) Nested For loop of depth 3
Performance counter stats for './llcout':

```
          4.480001      task-clock:u (msec)       #    0.928 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
                96      page-faults:u             #    0.021 M/sec
        10,873,047      cycles:u                  #    2.427 GHz
         8,361,108      instructions:u            #    0.77  insn per cycle
         1,645,073      branches:u                #  367.204 M/sec
            29,369      branch-misses:u           #    1.79% of all branches

       0.004825412 seconds time elapsed
```