Name : JSS SRAVAN CHANDRA

Professor : SURESH PURINI

Course : COMPILERS

13 November 2017

Design a Compiler for FlatB Programming Language

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

FlatB Programming Language Description:

1) Data Types used are Integers and 1D Array of Integers like int data, array[100];int sum; All the variables have to be declared in the declblock{....} before being used in the codeblock{...},they act like global variables. Multiple variables can be declared in the statement and each declaration statement ends with a semicolon. But same variable should not be declared twice.
2) This Language accepts Expressions Addition("+"), Subtraction("-") , Multiplication("*") , Division("/") with parenthesis.
3) For Loop can be defined in two ways

   for i = start, end { ..... } or

   for i = start, end, offset { ..... } here start, end , offset can be expressions

4) IF Conditional Statements can be in two forms

   if conditions { .... } or

   if conditions { .... } else { ..... }

5) Conditions are only comparative like "<", ">" ,"<=", ">=" , "==" , "!=" and those conditions can be combined with && (and condition) , || (or condition).

6) Print Statements can be in two forms

   print "something to print" [, variable]* (can give zero or more variables to print)

   println "something to print" [, variable]* (which gives a newline at the end)

7) Syntax for Read Statement is read variable [,variable]*

8) Conditional and Unconditional goto statements

   Any statement can have a label (where the syntax is "Label : " before any statement)

   Syntax for Conditional goto Statement is goto label if condition ;

   Syntax for Unconditional goto Statement is goto label ; (jumps to Label and execute from there)

Syntax and Semantic Analysis:

1) The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.

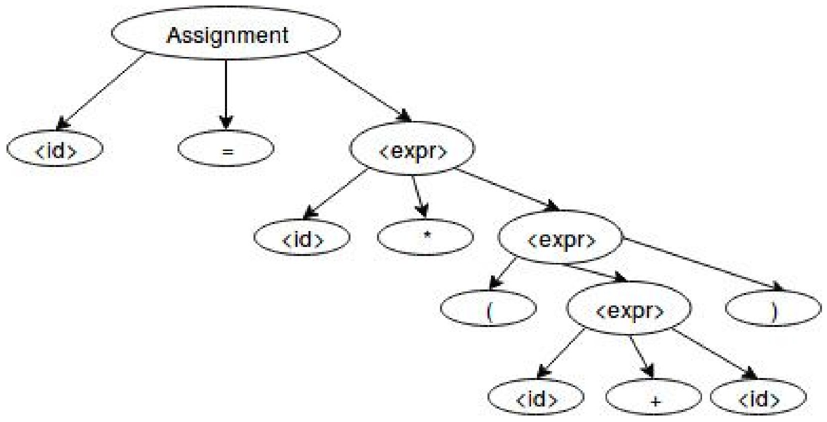
2) The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct and the parse tree is constructed using Context Free Grammar

   Example Context Free Grammar:

   Assignment : <id> = <expr>

```
<expr>          : <id> ‘*’ <expr>
                | <id> ‘+’ <expr>’
                | ‘(‘ <expr> ‘)’
                | <id>
```

Construction of parse tree for <id> = <id> * (<id> + <id>) using above CFG is



3) Semantic analysis checks whether the parse tree constructed follows the rules of language. Here I have checked is the variable is declared before its usage , and array limit, after construction of AST and this is done using Symbol Table.

Design of Abstract Syntax Tree:

After parsing we need to generate Abstract Syntax Tree so that we can easily generate intermediate code. Here for each node a class is defined with hierarchy.

Classes defined in my code with some hierarchy are

Class Program , Class Declaration_List, Class Code, Class Expression , Class Conditions,.....

Class Add:Class Expression{/*Add*/} …., Class Comparator : Conditions {/*Bool Compare*/}, Class For_loop:Class Code{/*For Loop*/}.., Class Goto:Class Code{/*Goto*/}, Class Print_Statement:Class Code{/*print*/} ….

Finally using these classes Abstract syntax tree is constructed using Bottom up parsing.

Visitor Design Pattern and how it is used:

The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class without disturbing the class objects. Here it is used to separate the algorithms from the data structures. After Construction of Abstract Syntax tree I used visitor design pattern to interpret the code.

The visit() methods declared in the Visitor base class using overloaded visit() method.Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy. The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass.

Design of Interpreter:

Interpreter is constructed using Visitor Design pattern using Abstract Syntax tree for the code , Symbol table to get or store the values of each variable . I checked whether variables declared or not , array limit here while interpreting.

Design of LLVM Code Generator:

After semantic analysis we generated an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code is generated using LLVM Code Generator.

Module in llvm is used to create a global level module.Variables are globally declared. Builder in llvm module is used to generate each instruction and Module->dump is used to generate the code.

For code

```
declblock{
        int x;
}
codeblock{
        x=2+3;
        println "x= " ,x;
}
```

Generated intermediate code is

```
; ModuleID = 'Sravan Chandra' /*new module declared*/
@x = global i32 0, align 4                                    /*declare variable x*/
@0 = private unnamed_addr constant [10 x i8] c"\22x= \22 %d\0A\00"
define void @main() {                                         /*main function declared*/
Start:                                                        /*declare a block*/
  store i32 5, i32* @x                                        /*assign value 5 to x*/
  %0 = load i32, i32* @x                                      /*load value of x*/
  %1 = call i32 @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @0, i32 0, i32 0), i32
%0)                                                    /*print statement*/
  ret void                                             /*end of the main dunction*/
}
declare i32 @printf(i8*, i32)
```

Performance Comparison:

TimeComparision	Interpreter	lli	llc
Bubble Sort	6.483s	0.614s	0.126s
Factorial	0.681s	0.043s	0.0198s
GCD	0.543s	0.030s	0.0161s

Bubble Sort	-> sort of 10 numbers 100000 times(test cases)
Factorial	-> find the factorial of 10 ,100000 times(test cases)
GCD	-> find gcd of 2 numbers ,100000 times(test cases)

Here Interpreter is included with Codegen too.

Time : Interpreter > lli>llc.

This is because our interpreter is coded in high level and lli converts it into bit code and interpretes from that whereas llc creates a executable file which the underlying processor can understand, optimized code so the time and number of instructions will be less than that of normal interpreter and lli.