“Compiler design for Selection statements in C++”

INTRODUCTION

We have built a compiler for the language C++. We have implemented various constructs and taken care of many different syntax requirements. We have also implemented the three address code generation and also the abstract syntax tree.For our optimization we are implementing constant propagation and constant folding.

ARCHITECTURE OF LANGUAGE

We have implemented the constructs for if, if-else ladder, switch and classes in this compiler. We have also taken care of nested functions and loops. We have taken care of if within switch and switch within if constructs as well.
Apart from that the normal syntax requirements like semicolon(;), brackets before every loop beginning and at the end of every loop is taken care of. Also the requirement to mention the data type when a variable is declared is also taken care of.

CONTEXT FREE GRAMMAR

start : include stmt ;

stmt : Assign ';' stmt | decl stmt | Function stmt | ClassStmt ';' stmt | ;

stmt1 : if stmt1 | if_else stmt1 | else_if stmt1 |switch stmt1 | coutstatement stmt1 | cinstatement stmt1 | Assign ';' stmt1 | decl stmt1 | ;

block : ';' | '{' stmt1 '}' ;

include : t_INCLUDE include | ;

if : t_IF '(' R_Exp ')' block ;
if_else: if t_ELSE block ;
else_if: if else_if1 ;
else_if1: t_ELSE t_IF '(' R_Exp ')' block else_if2;
else_if2: else_if1 | t_ELSE block ;

switch : t_SWITCH '(' t_ID ')' switch1;
switch1: ';' | '{' case '}'
case: case1 |case1 default ;

```
case1:  case2 case1 | ;
case2:  t_CASE t_NUM  ':' stmt1 | t_BREAK ';'  ;
default: t_DEFAULT  ':' stmt1 break ;
break : | t_BREAK';';

type : t_INT | t_FLOAT  | t_CHAR | t_DOUBLE  | t_VOID ;

decl : type var ';' ;
var : t_ID ',' var |t_ID ;

Assign : t_ID '='Exp {

ClassStmt : t_CLASS t_ID '{'  t_ACCESS ':' stmt  t_ACCESS ':' stmt '}'| t_CLASS t_ID ':'
t_ACCESS t_ID '{'  t_ACCESS ':' stmt  t_ACCESS ':' stmt '}' ;


Function: type t_ID '('ArgListOpt')' block  ;
ArgListOpt: ArgList|;
ArgList:  ArgList ',' Arg| Arg;
Arg: type var1 ;
var1 :t_ID;

coutstatement:t_COUT t_COUTOP t_COUTSTR t_COUTOP t_ENDL ';';

cinstatement: t_CIN t_CINOP t_ID ;

Exp   :  Exp'+' Exp | Exp'-' Exp| Exp'*' Exp| Exp'/' Exp| R_Exp| t_ID | t_NUM ;

R_Exp  : Exp'<' Exp | Exp'>' Exp | Exp t_LE Exp | Exp t_GE Exp | Exp t_EQ Exp | Exp
t_NE Exp | Exp t_OR Exp | Exp t_AND Exp ;
```

DESIGN STRATEGY

For the symbol table we thought of using a structure because there were so many different characteristics of a variable to implement. So we used an array of structures so that many variables could have the same structure.
For abstract syntax tree we assume that each function has different root nodes and all statements inside the function are stored in the tree as character arrays in the format {operator,operands..}.
For intermediate code we felt that the stack structure was best so that we could access the recently seen variables and then structure them properly. We generated the three address code based on this stack structure.

For the constant propagation and constant folding we had in mind the symbol table and since it involves substituting the constant values and then finding the answer we used the symbol table to obtain values of the variable to generate the final value of the output variable.

IMPLEMENTATION DETAILS

Symbol table is implemented using structure data type. The structure has the name, value, recent line number, the previous line it was mentioned on, the type, the scope and the property which is to say if its a class or ID and such.
Abstract syntax tree is implemented with the help of structure for the node which consists of left and right that are pointer to the node and character array that consists of the statement. Each statement is converted to a string and stored in the node.
Implementation of intermediate code generation involves stack which is implemented using array data type. We have labels to depict the control flow and temporary variable that help generate the three address code. The temporary variable is also put on the stack once it is generated.
We have executed one type of code optimization which is constant folding and constant propagation. In this we are using the symbol table to get the values of the variables and then generate output based on that. We are simply substituting the values and then storing the constant value of the temporary variables as well. The constant values of the temporary variables are stored in an array and then assigned to the variable pointing to the temporary variable in the symbol table. So symbol table has only values.

INSTRUCTIONS TO RUN THE CODE

flex remove.l
bison -dy head.y
gcc y.tab.c
a.exe

RESULT

In our final code we have generated a code which puts the variables into a symbol table and generates the three address code. It also optimizes it by substituting the constant values and then evaluating them.

SHORTCOMINGS

We still need to implement dead code elimination which would help make the code shorter than it is hence reducing evaluation time. Also we are not taking care of the scope of variables during evaluation. The variables are computed regardless of the scope. We were unable to implement the array data structure due to its complexity and time constraint.

CONCLUSIONS

In conclusion we would like to mention that we have a working code which takes care of the all the four phase of lexical analyzer, syntax analyzer, semantic analyzer and intermediate code generation completely.
Our optimization involves only two technique and those are constant propagation and constant folding. Our symbol table which keeps track of the various aspects of a variable aids in the optimization.
We have generated tokens using lex in the lexical analyzer phase and each token is initialized with T_tokenname. The yacc file takes care of all syntax evaluation. We have the symbol table generated as part of the semantic analyzer and the 3 address code and abstract syntax tree as part of the intermediate code generation phase.

FURTHER ENHANCEMENTS

We wish to implement more optimization techniques like the dead code elimination, loop optimization, packing temporaries, construction of DAG and live-variable analysis. Another goal is to implement data structures like arrays. We need to ensure that during evaluation of variables also the variables within the same scope are only being used.
With these enhancements we believe that the compiler will be more efficient and faster and than it is right now.