INTERMEDIATE CODE GENERATOR FOR A SUBSET OF C LANGUAGE USING LEX AND YACC TOOLS

Specifications

- Identifiers : alphabet (alphabet | digit)*
- Data types : int, unsigned int, boolean
- binary operators : +, -, *,/,@
- bitwise operators : | , & , ~ , ^
- logical operators : || , && , !
- Relational operators : == , != , < , <= , >, >=

Grammar used takes care of the precedence of operators

Control Structures

- if
- if-else
- switch
- while

Grammar used can parse programs of following structure:

global declarations : either simple declarations or one with assignment

```
int main( )
{
        declarations;

        assignments;

        if (expression)
        {
                statements;
        }
        else
        {
                statements;
        }
        while (expression)
        {
                statements;
        }
        switch ( ID )
        {
                case 1 : statements;
                         break;
                case 2 : statements;
                default : statements;
        }
}
```

Compile Instructions

lex file containing regular expressions for recognising tokens : c.l
yacc file containing grammar rules : c.y

```
$lex    c.l
$yacc   c.y
```

```
$gcc    y.tab.c    -ll    -ly
```

Lex Specifications

Whenever a lexeme matches any patterns listed in lex file corresponding token is returned in the lex file.

eg. “>=” return GE

For recognising identifier

```
alpha       [a-zA-Z]
digit       [0-9]
```

```
{alpha}({alpha}|{digit})*       return      ID
```

Note All the tokens that are being returned by lexical analyser needs to be defined in yacc file

eg. %token if else break switch

For ignoring single line and multiline comments,no action is required if lexeme matches that pattern:

```
\\.*                ;
\/*(.*\n)*.*\*\/    ;
```

Now it's time for Yacc.....

Declaration part of yacc program consists of definition for tokens,left associative and right associative operators along with some additional code enclosed within pair of %{

```
%{
        code
%}
```

included code is copied as it is to generated c file.

```
%token      IF ELSE PRINTF SWITCH CASE BREAK
%left       '<' '>' LE GE EQ NE LT GT
%right      ASSIGN
%left       AND OR
```

code can start with either global declarations or with main function.

```
start :   declaration start
      |   main start
      |
      ;
```

where main function can have structure like

```
main  : INT   MAIN '(' ')' block
      ;
```

```
//set of statements enclosed within a pair of braces
block : '{' stmts '}'
      ;
```

A single statement can take any of the following forms :

```
stmt  :  assignment ';'
         | declaration
         | shorthand ';'
         | while
         | if
         | switch
         ;
```

Now it's time to have a look at precedence of operators.
In grammar listed below operators having lower precedence are written before the ones having higher precedence.This ensures that expressions corresponding to higher precedence operators will be evaluated first.

```
expr  : expr AND E
         | expr OR  E
         | E
         ;

E        : E  '|'  F
         | E  '^' F
         | E  '&' F
         | E  '~' F
         | F
         ;

F        : F    LE     G
         | F    GE     G
         | F    EQ     G
         | F    NE     G
         | F    LT     G
         | F    GT     G
         | G
         ;
```

```
G       : G  '+' H
        | G  '-'  H
        | H
        ;

H       : H  '*'  I
        | H  '/'  I
        | I
        ;

I       : J  '@' I
        | J
        ;

J       : '(' expr ')'
        | ID
        | NUM
        ;
```

Similarly there are so many grammar rules for other type of statements including while,if-else, switch,etc.For detailed description refer to the code attached to this file.

Time to move on to Intermediate Code Generation

Mainly there are 2 approaches to generate intermediate code from the grammar.

- Using attributes of a symbol by expressions of the form $$.addr=$1.addr '+' $2.addr and other type of expressions involving $ at each node of the tree.

- Using a stack of grammar symbols onto which yytext is pushed and popped whenever needed. Another stack is also used to keep track of the labels to be generated when dealing with control flow statements.

In this project I've followed second approach.

Let's start with a simple instruction

```
G : G '+' H
```

value corresponding to G on RHS obtained from lower level nodes needs to be pushed onto the stack.And after obtaining value corresponding to H code needs to be generated.

Specifying action part.....

```
G       : G    '+'{push();}    H{expressions();}
```

push() function will simply copy the content of the string yytext to the top of the stack.

```
//push current literal on the stack
push()
{
	strcpy(st[++top],yytext);
	yytext[0]='\0';
}
```

expressions() function will generate intermediate code by printing out the top 3 elements of the stack in order,after which top will be decremented by 2.

```
expressions()
{
	sprintf(temp,"t%d",i);
	sprintf(table[linecount],"%s = %s %s %s",temp,st[top-2],st[top-1],st[top]);
	linecount++;
	top-=2;
	strcpy(st[top],temp);
	i++;
}
```

Similar approach is followed for assignment operations,shorthand notation,etc.

Let's have a look at intermediate code generation for while statement to get an idea of how to deal with label and gotos.

```
while : WHILE {while_label1();} '(' expr ')'{while_label2();} whilefactor
	;

//to eliminate shift/reduce conflict
whilefactor : block{while_label3();}
		| stmt{while_label3();}
		;
```

while_label1() will generate label at the start of the loop,where jump has to be made after each iteration.
while_label2() will generate jump using goto based on condition inside a pair of parentheses.
while_label3() will generate label at the end of loop where control needs to be transferred when condition becomes false.

Forward Referencing is resolved using stack of labels . Whenever an expression requires a goto statement, one label is generated and pushed onto the stack . When target statement is encountered, label is popped out from the stack and assigned.

```
while_label1()
{
	lnum++;
	sprintf(table[linecount],"L%d:",lnum);
	linecount++;
	label[ltop++] = lnum++;
}
```

```
while_label2()
{
	sprintf(temp,"t%d",i);
	sprintf(table[linecount],"%s = not %s",temp,st[top]);
	linecount++;
	sprintf(table[linecount],"if %s goto L%d",temp,lnum);
	linecount++;
	label[ltop++] = lnum;
	i++;
}
```

```
while_label3()
{
	sprintf(table[linecount],"goto L%d",label[ltop-2]);
	linecount++;
	sprintf(table[linecount],"L%d:",label[ltop-1]);
	linecount++;
	ltop = ltop-2;
}
```

Keeping track of declarations using symbol table

Symbol table is used which will keep track of all the declared ID's.

If any identifer is declared before it will report and error stating that ID was previously declared, otherwise that ID will be inserted into the table.

If a variable is used without being declared then also error will be reported.
All this is done using 2 functions listed below :

```
//insert each declared variable in the table,and give error if already declared
insert_into_symbol_table(char *str)
{
	int i=0;
	for(;i<size;i++)
	{
		if(strcmp(symbolTable[i],str)==0)
		{
			printf("Error line(%d): %s is redeclared\n",yylineno, str);
			exit(0);
		}
	}
	printf("%s added to symbol table\n",str);
	strcpy(symbolTable[size++],str);
}
```

```
//checks whether an identifier is declared or not by scanning the symbol table
check_if_declared(char *str)
{
	int i=0;
	for(;i<size;i++)
	{
		if(strcmp(symbolTable[i],str)==0)
			break;
	}
	if(i==size)
	{
		printf("Error line(%d): %s not declared \n",yylineno, str);
		exit(0);
	}
}
```

This is all about the briefing of the project.
For detailed description refer to the code attached.

INPUT FILE

```
int main()
{
   int x,y,z=4;
   int a;
   x=y;
   y=7;
   z=3;
   z*=5;
   while(y+z<=20)
   {
        y=y+1;
        z=x+2*5+(4+7)*(5/y);
        switch(a)
        {
                case 1 : y=1;
                                break;
                case 2 : y=2;
                default : y=x;
        }
   }
   a=5;
   if(x<2)
   {
     x=x+5;
   }
   else
   {
     x=x-5;
   }

}
```

OUTPUT

```
x added to symbol table
y added to symbol table
z added to symbol table
a added to symbol table

Parsing completed successfully

Enter any key to generate intermediate code....
h

Intermediate code :
0       z = 4
1       x = y
2       y = 7
3       z = 3
4       z = z * 5
5       t0 = y + z
6       t1 = t0 <= 20
7       t2 = not t1
8       if t2 goto 28
9       t3 = y + 1
10      y = t3
11      t4 = 2 * 5
12      t5 = x + t4
13      t6 = 4 + 7
14      t7 = 5 / y
15      t8 = t6 * t7
16      t9 = t5 + t8
17      z = t9
18      if z != 1 goto 22
19
20      y = 1
21      goto 27
22      if z != 2 goto 26
23
24      y = 2
25      goto
26      y = x
27      goto 5
28      a = 5
29      t12 = x < 2
30      t13 = not t12
31      if t13 goto 35
```

```
32      t14 = x + 5
33      x = t14
34      goto 37
35      t15 = x - 5
36      x = t15
```