**Intermediate Code Generation**

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language. The given program in a source language is converted into an equivalent program in an intermediate language by the intermediate code generator. Intermediate codes are machine independent codes.

Roles of Intermediate code generation phase are :

* It acts as a glue between front-end and backend (or source and machine codes).
* If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
* The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
* Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
* Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.
* Intermediate code generator lowers abstraction from source level.

The intermediate code representation are

* Graphical representation e.g. Abstract Syntax Tree(AST) , DAGS
* Postfix notations
* Three Address codes

**Three Address Code**

**Three address code** is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

**General representation –**

**a = b op c**

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

* Assignments x=y op z ;x = op y.
* Copy x = y.
* Unconditional jump goto L.
* Conditional jumps if x relop y goto L.
* Parameters param x.
* Function call y = call p

**Implementation of Three Address Code –**  
There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

**Code snippets with explanation**

Snippets from **scanner.l**

void insertSTnest(char \*s, int nest)

{

if(lookupST(s) && ST[lookupST(s)].nestval != 9999)

{

int pos = 0;

int value = hash(s);

for (int i = value + 1 ; i!=value ; i = (i+1)%1001)

{

if(ST[i].length == 0)

{

pos = i;

break;

}

}

strcpy(ST[pos].name,s);

strcpy(ST[pos].class,"Identifier");

ST[pos].length = strlen(s);

ST[pos].nestval = nest;

ST[pos].params\_count = -1;

ST[pos].lineno = yylineno;

}

else

{

for(int i = 0 ; i < 1001 ; i++)

{

if(strcmp(ST[i].name,s)==0 )

{

ST[i].nestval = nest;

}

}

}

}

void insertSTparamscount(char \*s, int count1)

{

for(int i = 0 ; i < 1001 ; i++)

{

if(strcmp(ST[i].name,s)==0 )

{

ST[i].params\_count = count1;

}

}

}

int checkscope(char \*s)

{

int flag = 0;

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].name,s)==0)

{

if(ST[i].nestval > currnest)

{

flag = 1;

}

else

{

flag = 0;

break;

}

}

}

if(!flag)

{

return 1;

}

else

{

return 0;

}

}

void deletedata (int nesting)

{

for(int i = 0 ; i < 1001 ; i++)

{

if(ST[i].nestval == nesting)

{

ST[i].nestval = 99999;

}

}

}

int duplicate(char \*s)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].name,s)==0)

{

if(ST[i].nestval == currnest)

{

return 1;

}

}

}

return 0;

}

Snippets from **parser.y**

variable\_declaration\_identifier

: identifier {if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi

| array\_identifier {if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi;

call

: identifier '('{

if(!check\_declaration(curid, "Function"))

{ printf("Function not declared"); exit(0);}

insertSTF(curid);

strcpy(currfunccall,curid);

if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')

{

$$ = 1;

}

else

$$ = -1;

call\_params\_count=0;

}

arguments ')'

{ if(strcmp(currfunccall,"printf"))

{

if(getSTparamscount(currfunccall)!=call\_params\_count)

{

yyerror("Number of arguments in function call doesn't match number of parameters");

exit(8);

}

}

callgen();

};

void codegen()

{

strcpy(temp,"t");

char buffer[100];

itoa(count,buffer,10);

strcat(temp,buffer);

printf("%s = %s %s %s\n",temp,s[top-2].value,s[top-1].value,s[top].value);

top = top - 2;

strcpy(s[top].value,temp);

count++;

}

void codegencon()

{

strcpy(temp,"t");

char buffer[100];

itoa(count,buffer,10);

strcat(temp,buffer);

printf("%s = %s\n",temp,curval);

push(temp);

count++;

}

int isunary(char \*s)

{

if(strcmp(s, "--")==0 || strcmp(s, "++")==0)

{

return 1;

}

return 0;

}

void genunary()

{

char temp1[100], temp2[100], temp3[100];

strcpy(temp1, s[top].value);

strcpy(temp2, s[top-1].value);

if(isunary(temp1))

{

strcpy(temp3, temp1);

strcpy(temp1, temp2);

strcpy(temp2, temp3);

}

strcpy(temp, "t");

char buffer[100];

itoa(count, buffer, 10);

strcat(temp, buffer);

count++;

if(strcmp(temp2,"--")==0)

{

printf("%s = %s - 1\n", temp, temp1);

printf("%s = %s\n", temp1, temp);

}

if(strcmp(temp2,"++")==0)

{

printf("%s = %s + 1\n", temp, temp1);

printf("%s = %s\n", temp1, temp);

}

top = top -2;

}

**Explanation**

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like insertSTnest(), insertSTparamscount(), checkscope(), deletedata(), duplicate() ​etc., in order to check the semantics​.​In the production rules of the grammar semantic actions are written and these are performed by the functions listed above. Along with semantic actions SDT also included function to generate the 3 address code.

**Implementation:**

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

A. The Regex for Identifiers  
B. Multiline comments should be supported C. Literals  
D. Error Handling for Incomplete String  
E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.  
The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type , value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not. Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we made use of explicit stack. Whenever we came across an operator, operand or constant we pushed it to stack. Whenever reduction occurred (Since LALR(1) parser is bottom up parser it evaluates SDT when reduction occurs) codegen( ) function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack, after that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation. Similarly functions like labels were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below :

1. codegen( ) : This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e x = y op z.
2. codegencon():Thisfunctionisespeciallywrittenforreductionsofexpression involving constants since its 3 address code is x op z.
3. isunary( ) : This function checks if the operator is an unary operator like ‘++’. If so it returns true else false.
4. genunary( ) : This function is specifically designed to generate 3 address code for unary operations. It makes use of isuanary function mentioned above. E.g. if a = i++ then it converts into t0 = i + 1, a = t0.

**Testcases with outputs**

**Testcase 1 - Valid Testcase: O**​**perator, Delimiters, Assignments, Nested Conditional Statements, for and while loops**

#include <stdio.h>

int myfunc(int a,int b)

{

return a+b;

}

void main()

{

int a,b,i;

while(a<3)

{

a = a+b;

for(i=0;i<b;i++)

{

b++;

myfunc(a,b);

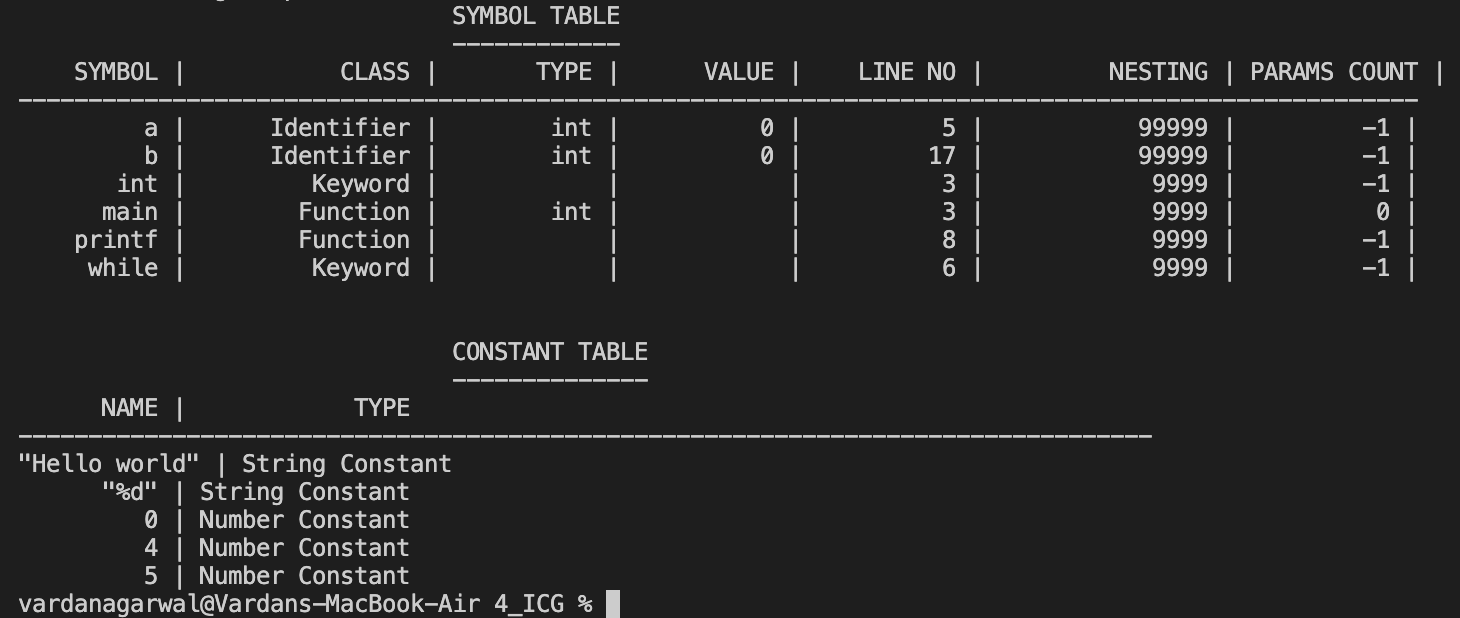
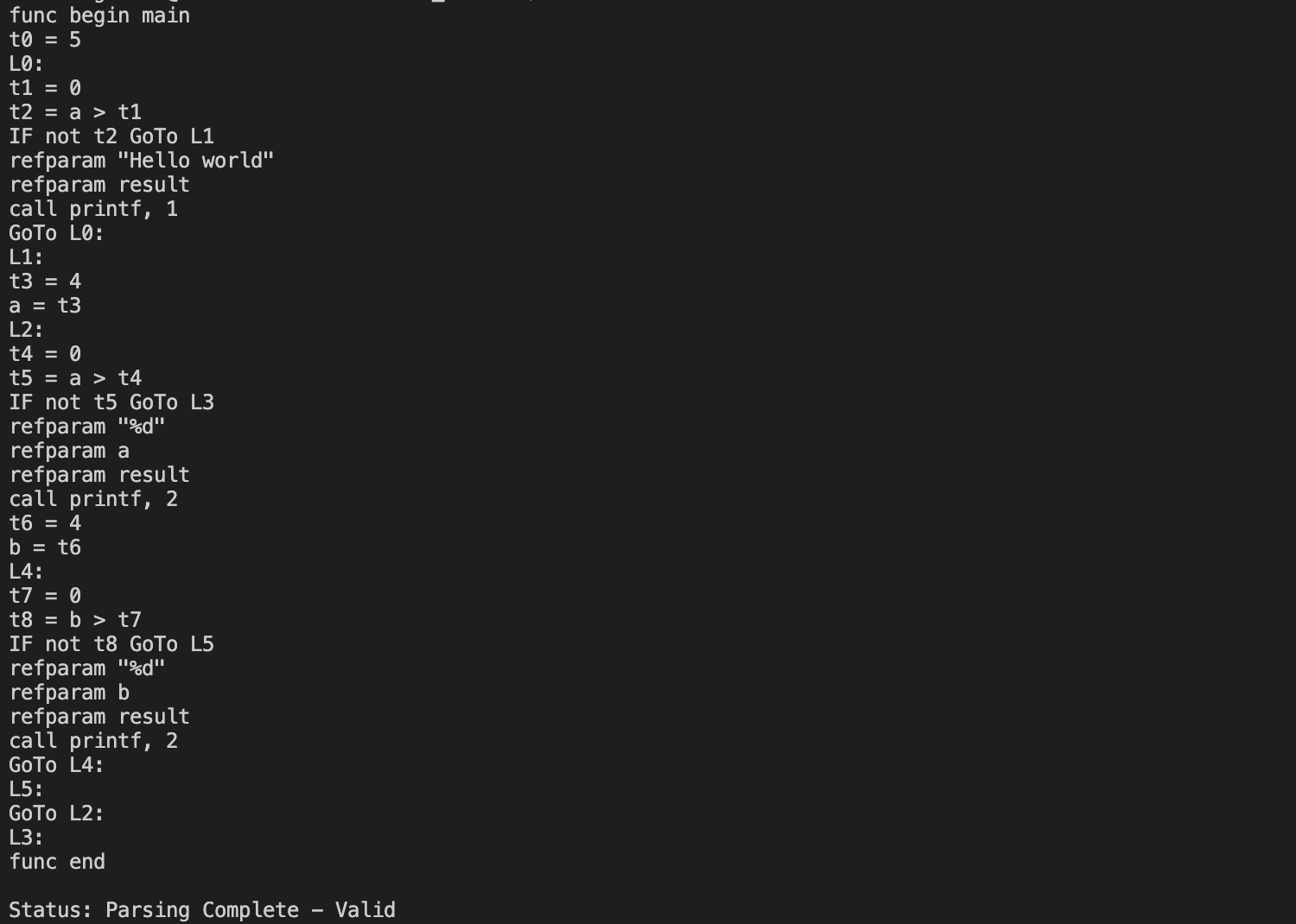
}

a++;

}

}

**Output: -**



**Testcase 2 - Inalid Testcase:**

// Implicit Error that our Language doesn't support

#include<stdio.h>

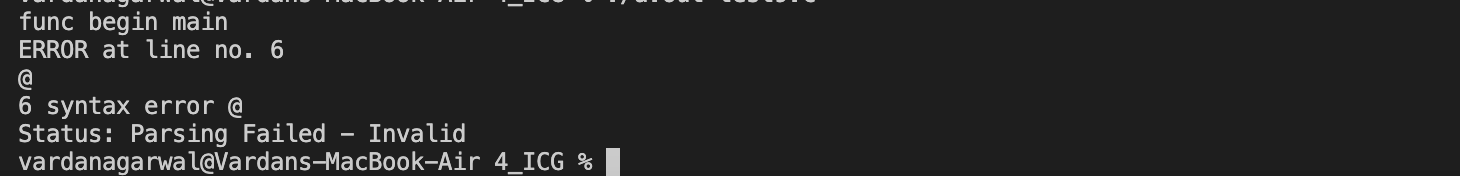
int main() {

char @hello;

@hello = 'c';

}

**Output: -**



**Testcase 3 - Inalid Testcase:**

#include<stdio.h>

int square(int a, int b)

{

int b = 2;

return b;

}

int main()

{

int num = 2;

int num2;

square(num,num);

//printf("Square of %d is %d", num, square2(5));

return 0;

}

**Output: -**

