

National Institute of Technology Karnataka Surathkal

**Project 3: Intermediate Code Generation for C Language**

**Date:**13/04/2018

**Submitted To:** Ms.Sushmita G

**Group Members:**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No | Name | E-mail | Roll Number |
| 1 | Naladala Indukala | [15co230.indukala@nitk.edu.in](mailto:15co230.indukala@nitk.edu.in) | 15CO230 |
| 2 | R. Aparna | [15co236.aparna@nitk.edu.in](mailto:15co236.aparna@nitk.edu.in) | 15CO236 |

# **Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Title** | **Page No.** |
| 1 | Introduction | 3 |
| 2 | Design of Programs | 5 |
| 3 | Implementation Explanation | 26 |
| 4 | Back-patching | 30 |
| 5 | Test Cases | 31 |
| 6 | Conclusion and Future Work | 40 |
| 7 | References | 40 |

**Introduction**

**Intermediate Code Generation**

The final goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code which can run on the computer. This does not mean that we need to translate directly from the high-level abstract syntax to machine code. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code. We will generate intermediate code using translation functions for each syntactic category, similar to the functions we used for interpretation and type checking.

Intermediate code generator receives input from semantic analyser in the form of annotated syntax tree. Syntax tree is converted to its linear representation. The resulting three address code has at most three address locations in each instruction. So the three address code (3AC) of an expression like a\*(b+c)-d is generated by dividing into sub-expressions. We have represented our three address code as quadruples. Each instruction in quadruple has four fields, namely, operator, arg1, arg2 and result. The other ways to represent 3AC are triplets (has three fields: operator, arg1, arg2) and indirect triplets. Triplets use position of expression to store result of sub-expression while indirect triplets use pointers. Our ICG generates 3AC for:

* Expressions
* Assignments
* If Else
* While
* For
* Do while
* Functions

Some benefits of using a machine-independent intermediate form are:

* Retargeting is facilitated**:** A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
* A machine-independent code optimizer can be applied to the intermediate representation.

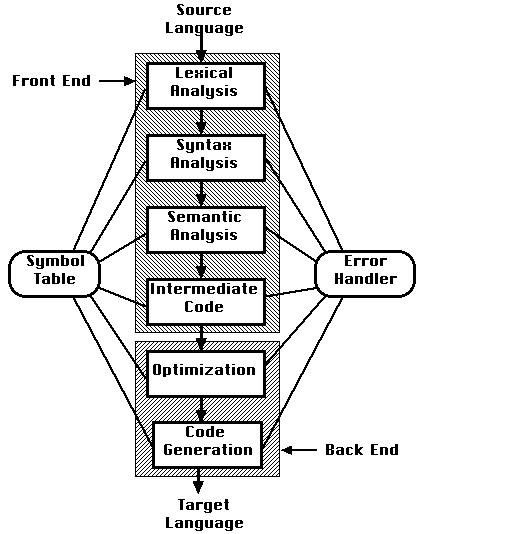
Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

* [Instruction selection](https://en.wikipedia.org/wiki/Instruction_selection): Which instructions to use.
* [Instruction scheduling](https://en.wikipedia.org/wiki/Instruction_scheduling): In which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on [pipelined](https://en.wikipedia.org/wiki/Instruction_pipeline) machines.
* [Register allocation](https://en.wikipedia.org/wiki/Register_allocation): The allocation of [variables](https://en.wikipedia.org/wiki/Variable_(programming)) to [processor registers](https://en.wikipedia.org/wiki/Processor_register)

Decisions in intermediate code generation design affect the speed and efficiency of the compiler.

Important Intermediate code properties:

* Ease of generation
* Ease of manipulation
* Procedure size
* Freedom of expression
* Level of abstraction



**C Program**

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into account. A file called lex.yy.c is generated, which when executed, recognizes the tokens present in input C program. The yacc script parses the C code given as input and specifies the syntactic and semantic errors (if any) or tells the user that the parsing has been successfully completed. It then generates 3AC.

**Design of Programs**

**Codes**

**Lex Code : (scanner.l file)**

|  |
| --- |
| %{ |
|  |

|  |
| --- |
| int count=0; |
|  |

|  |
| --- |
| %} |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| alpha [A-Za-z] |
|  |

|  |
| --- |
| digit [0-9] |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| %% |
|  |

|  |
| --- |
| [ \t] ; |
|  |

|  |
| --- |
| \n {yylineno++;} |
|  |

|  |
| --- |
| "{" {open1(); count++; return '{';} |
|  |

|  |
| --- |
| "}" {close1(); count--; return '}';} |
|  |

|  |
| --- |
| int {yylval.ival = INT; return INT;} |
|  |

|  |
| --- |
| float {yylval.ival = FLOAT; return FLOAT;} |
|  |

|  |
| --- |
| void {yylval.ival = VOID; return VOID;} |
|  |

|  |
| --- |
| else {return ELSE;} |
|  |

|  |
| --- |
| do return DO; |
|  |

|  |
| --- |
| if return IF; |
|  |

|  |
| --- |
| ^"#include ".+ return PREPROC; |
|  |

|  |
| --- |
| while return WHILE; |
|  |

|  |
| --- |
| for return FOR; |
|  |

|  |
| --- |
| return return RETURN; |
|  |

|  |
| --- |
| printf return PRINT; |
|  |

|  |
| --- |
| {alpha}({alpha}|{digit})\* {yylval.str=strdup(yytext); return ID;} |
|  |

|  |
| --- |
| {digit}+ {yylval.str=strdup(yytext);return NUM;} |
|  |

|  |
| --- |
| {digit}+\.{digit}+ {yylval.str=strdup(yytext); return REAL;} |
|  |

|  |
| --- |
| "<=" return LE; |
|  |

|  |
| --- |
| ">=" return GE; |
|  |

|  |
| --- |
| "==" return EQ; |
|  |

|  |
| --- |
| "!=" return NEQ; |
|  |

|  |
| --- |
| "&&" return AND; |
|  |

|  |
| --- |
| "||" return OR; |
|  |

|  |
| --- |
| \/\/.\* ; |
|  |

|  |
| --- |
| \/\\*(.\*\n)\*.\*\\*\/ ; |
|  |

|  |
| --- |
| \".\*\" return STRING; |
|  |

|  |
| --- |
| . return yytext[0]; |
|  |

|  |
| --- |
| %% |
|  |

**Yacc Code (parser.y)**

|  |
| --- |
| %{ |
|  |

|  |
| --- |
| #include <stdio.h> |
|  |

|  |
| --- |
| #include <stdlib.h> |
|  |

|  |
| --- |
| int function=0; |
|  |

|  |
| --- |
| #include "symbol.c" |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int i=1,label1[20]; |
|  |

|  |
| --- |
| int stack[100],index1=0,end[100],arr[10];  int dom,roo,top=0,label[20],name=0,ltop=0; |
|  |

|  |
| --- |
| char st1[100][10]; |
|  |

|  |
| --- |
| char ig[2]="0"; |
|  |

|  |
| --- |
| char temp[2]="t"; |
|  |

|  |
| --- |
| void yyerror(char \*s); |
|  |

|  |
| --- |
| int printline(); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void for3() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto L%d\n",label[ltop-3]); |
|  |

|  |
| --- |
| printf("L%d:\n",label[ltop-1]); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void flow\_1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| strcpy(temp,"t"); |
|  |

|  |
| --- |
| strcat(temp,ig); |
|  |

|  |
| --- |
| printf("%s = not %s\n",temp,st1[top]); |
|  |

|  |
| --- |
| printf("if %s goto L%d\n",temp,name); |
|  |

|  |
| --- |
| ig[0]++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void flow\_2() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| printf("goto L%d\n",name); |
|  |

|  |
| --- |
| printf("L%d: \n",label[ltop--]); |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void flow\_3() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("L%d:\n",label[ltop--]); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void while1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| printf("L%d:\n",name); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void while2() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| strcpy(temp,"t"); |
|  |

|  |
| --- |
| strcat(temp,ig); |
|  |

|  |
| --- |
| printf("%s = not %s\n",temp,st1[top--]); |
|  |

|  |
| --- |
| printf("if %s goto L%d\n",temp,name); |
|  |

|  |
| --- |
| ig[0]++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void while3() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int y=label[ltop--]; |
|  |

|  |
| --- |
| printf("goto L%d\n",label[ltop--]); |
|  |

|  |
| --- |
| printf("L%d:\n",y); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void dowhile1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| printf("L%d:\n",name); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void dowhile2() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("if %s goto L%d\n",st1[top--],label[ltop--]); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void close1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| index1--; |
|  |

|  |
| --- |
| end[stack[index1]]=1; |
|  |

|  |
| --- |
| stack[index1]=0; |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void for1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| printf("L%d:\n",name); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void for2() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| strcpy(temp,"t"); |
|  |

|  |
| --- |
| strcat(temp,ig); |
|  |

|  |
| --- |
| printf("%s = not %s\n",temp,st1[top--]); |
|  |

|  |
| --- |
| printf("if %s goto L%d\n",temp,name); |
|  |

|  |
| --- |
| ig[0]++; |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| printf("goto L%d\n",name); |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| name++; |
|  |

|  |
| --- |
| printf("L%d:\n",name); |
|  |

|  |
| --- |
| label[++ltop]=name; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void for4() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto L%d\n",label[ltop]); |
|  |

|  |
| --- |
| printf("L%d:\n",label[ltop-2]); |
|  |

|  |
| --- |
| ltop-=4; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void push(char \*a) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| strcpy(st1[++top],a); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void icg() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| strcpy(temp,"t"); |
|  |

|  |
| --- |
| strcat(temp,ig); |
|  |

|  |
| --- |
| printf("%s = %s %s %s\n",temp,st1[top-2],st1[top-1],st1[top]); |
|  |

|  |
| --- |
| top-=2; |
|  |

|  |
| --- |
| strcpy(st1[top],temp); |
|  |

|  |
| --- |
| ig[0]++; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void open1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| stack[index1]=i; |
|  |

|  |
| --- |
| i++; |
|  |

|  |
| --- |
| index1++; |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void icg\_1() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("%s = %s\n",st1[top-2],st1[top]); |
|  |

|  |
| --- |
| top-=2; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| %} |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| %token<ival> INT FLOAT VOID |
|  |

|  |
| --- |
| %token<str> ID NUM REAL LE GE EQ NEQ AND OR |
|  |

|  |
| --- |
| %token WHILE IF RETURN PREPROC STRING PRINT FUNCTION DO ARRAY ELSE FOR |
|  |

|  |
| --- |
| %left LE GE EQ NEQ AND OR '<' '>' |
|  |

|  |
| --- |
| %right '=' |
|  |

|  |
| --- |
| %left '+' '-' |
|  |

|  |
| --- |
| %left '\*' '/' |
|  |

|  |
| --- |
| %type<str> assignment1 val '=' '+' '-' '\*' '/' E T F |
|  |

|  |
| --- |
| %type<ival> Type |
|  |

|  |
| --- |
| %union { |
|  |

|  |
| --- |
| int ival; |
|  |

|  |
| --- |
| char \*str; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| %% |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| start : Function start |
|  |

|  |
| --- |
| | PREPROC start |
|  |

|  |
| --- |
| | Declaration start |
|  |

|  |
| --- |
| | |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| function\_call : ID '(' ')' ';' { |
|  |

|  |
| --- |
| int k=lookup\_func($1); |
|  |

|  |
| --- |
| if(k==-1) |
|  |

|  |
| --- |
| printf(" \nUndefined function : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(number\_param(k)!=0) |
|  |

|  |
| --- |
| printf("\nNumber of parameters is invalid : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto F%d\n",k); |
|  |

|  |
| --- |
| printf("M%d:\n",k); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | ID '(' constant\_list ')' ';' { |
|  |

|  |
| --- |
| int k=lookup\_func($1); |
|  |

|  |
| --- |
| if(k==-1) |
|  |

|  |
| --- |
| printf(" \nUndefined function : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(number\_param(k)!=2) |
|  |

|  |
| --- |
| printf("\nNumber of parameters is invalid : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto F%d\n",k); |
|  |

|  |
| --- |
| printf("M%d:\n",k); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | ID '(' constant\_list2 ')' ';' { |
|  |

|  |
| --- |
| int k=lookup\_func($1); |
|  |

|  |
| --- |
| if(k==-1) |
|  |

|  |
| --- |
| printf(" \nUndefined function : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(number\_param(k)!=3) |
|  |

|  |
| --- |
| printf("\nNumber of parameters is invalid : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto F%d\n",k); |
|  |

|  |
| --- |
| printf("M%d:\n",k); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | ID '(' argument ')' ';' { |
|  |

|  |
| --- |
| int k=lookup\_func($1); |
|  |

|  |
| --- |
| if(k==-1) |
|  |

|  |
| --- |
| printf(" \nUndefined function : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(number\_param(k)!=1) |
|  |

|  |
| --- |
| printf("\nNumber of parameters is invalid : Line %d\n",printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("goto F%d\n",k); |
|  |

|  |
| --- |
| printf("M%d:\n",k); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| argument: val | ID ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| constant\_list: argument ',' argument ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| constant\_list2: argument ',' argument ',' argument ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| Function : Type ID '('')' {if(strcmp($2,"main")!=0) {printf("F%d:\n",function);function++;} else printf("M:\n");} CompoundStmt { |
|  |

|  |
| --- |
| if(strcmp($2,"main")!=0) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_func($2,function-1,0); |
|  |

|  |
| --- |
| printf("goto M%d\n",function-1); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if ($1!=returntype\_func(dom)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") && strcmp($2,"getchar") && strcmp($2,"puts") && strcmp($2,"putchar"))) |
|  |

|  |
| --- |
| printf("Error : Type mismatch in redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert($2,FUNCTION,nesting()); |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | Type ID '(' Type ID ')' {if(strcmp($2,"main")!=0) {printf("F%d:\n",function);function++;} else printf("M:\n");} CompoundStmt { |
|  |

|  |
| --- |
| if(strcmp($2,"main")!=0) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_func($2,function-1,1); |
|  |

|  |
| --- |
| printf("goto M%d\n",function-1); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if ($1!=returntype\_func(dom)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") && strcmp($2,"putchar"))) |
|  |

|  |
| --- |
| printf("Error : Type mismatch in redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert($2,FUNCTION,nesting()); |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | Type ID '(' Type ID ',' Type ID ')' {if(strcmp($2,"main")!=0) {printf("F%d:\n",function);function++;} else printf("M:\n");}CompoundStmt { |
|  |

|  |
| --- |
| if(strcmp($2,"main")!=0) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_func($2,function-1,2); |
|  |

|  |
| --- |
| printf("goto M%d\n",function-1); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if ($1!=returntype\_func(dom)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") && strcmp($2,"putchar"))) |
|  |

|  |
| --- |
| printf("Error : Type mismatch in redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert($2,FUNCTION,nesting()); |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | Type ID '(' Type ID ',' Type ID ',' Type ID ')' {if(strcmp($2,"main")!=0) {printf("F%d:\n",function);function++;} else printf("M:\n");} CompoundStmt { |
|  |

|  |
| --- |
| if(strcmp($2,"main")!=0) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_func($2,function-1,3); |
|  |

|  |
| --- |
| printf("goto M%d\n",function-1); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if ($1!=returntype\_func(dom)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") && strcmp($2,"putchar"))) |
|  |

|  |
| --- |
| printf("Error : Type mismatch in redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert($2,FUNCTION,nesting()); |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| Type : INT |
|  |

|  |
| --- |
| | FLOAT |
|  |

|  |
| --- |
| | VOID |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| CompoundStmt : '{' StmtList '}' |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| StmtList : StmtList stmt |
|  |

|  |
| --- |
| | |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| stmt : Declaration |
|  |

|  |
| --- |
| | if |
|  |

|  |
| --- |
| | while |
|  |

|  |
| --- |
| | dowhile |
|  |

|  |
| --- |
| | for |
|  |

|  |
| --- |
| | function\_call |
|  |

|  |
| --- |
| | RETURN val ';' { |
|  |

|  |
| --- |
| if(!(strspn($2,"0123456789")==strlen($2))) |
|  |

|  |
| --- |
| storereturn(dom,FLOAT); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| storereturn(dom,INT); dom++; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | RETURN ';' {storereturn(dom,VOID); dom++;} |
|  |

|  |
| --- |
| | ';' |
|  |

|  |
| --- |
| | PRINT '(' STRING ')' ';' |
|  |

|  |
| --- |
| | CompoundStmt |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| dowhile : DO {dowhile1();} CompoundStmt WHILE '(' E ')' {dowhile2();} ';' |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| for : FOR '(' E {for1();} ';' E {for2();}';' E {for3();} ')' CompoundStmt {for4();} |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if : IF '(' E ')' {flow\_1();} CompoundStmt {flow\_2();} else |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| else : ELSE CompoundStmt {flow\_3();} |
|  |

|  |
| --- |
| | |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| while : WHILE {while1();}'(' E ')' {while2();} CompoundStmt {while3();} |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| assignment1 : ID {push($1);} '=' {strcpy(st1[++top],"=");} E {icg\_1();} |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int sct=returnscope($1,stack[index1-1]); |
|  |

|  |
| --- |
| int type=returntype($1,sct); |
|  |

|  |
| --- |
| if((!(strspn($5,"0123456789")==strlen($5))) && type==258 && roo==0) |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| if(!lookup($1)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int currscope=stack[index1-1]; |
|  |

|  |
| --- |
| int scope=returnscope($1,currscope); |
|  |

|  |
| --- |
| if((scope<=currscope && end[scope]==0) && !(scope==0)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| check\_scope\_update($1,$5,currscope); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| | ID ',' assignment1 { |
|  |

|  |
| --- |
| if(lookup($1)) |
|  |

|  |
| --- |
| printf("\nUndeclared Variable %s : Line %d\n",$1,printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | val ',' assignment1 |
|  |

|  |
| --- |
| | ID { |
|  |

|  |
| --- |
| if(lookup($1)) |
|  |

|  |
| --- |
| printf("\nUndeclared Variable %s : Line %d\n",$1,printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | val |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| val : NUM |
|  |

|  |
| --- |
| | REAL |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| E : E '+'{strcpy(st1[++top],"+");} T{icg();} |
|  |

|  |
| --- |
| | E '-'{strcpy(st1[++top],"-");} T{icg();} |
|  |

|  |
| --- |
| | T |
|  |

|  |
| --- |
| | ID {push($1);} EQ {strcpy(st1[++top],"==");} E {icg();} |
|  |

|  |
| --- |
| | ID {push($1);} AND {strcpy(st1[++top],"&&");} E {icg();} |
|  |

|  |
| --- |
| | ID {push($1);} OR {strcpy(st1[++top],"||");} E {icg();} |
|  |

|  |
| --- |
| | ID {push($1);} '<' {strcpy(st1[++top],"<");} E {icg();} |
|  |

|  |
| --- |
| | ID {push($1);} '>' {strcpy(st1[++top],">");} E {icg();} |
|  |

|  |
| --- |
| | ID {push($1);} '=' {strcpy(st1[++top],"||");} E {icg\_1();} |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| T : T '\*'{strcpy(st1[++top],"\*");} F {icg();} |
|  |

|  |
| --- |
| | T '/'{strcpy(st1[++top],"/");} F {icg();} |
|  |

|  |
| --- |
| | F |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| F :'(' E ')' {$$=$2;} |
|  |

|  |
| --- |
| | ID {push($1);roo=1;} |
|  |

|  |
| --- |
| | val {push($1);} |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| Declaration : Type ID ';' |
|  |

|  |
| --- |
| { if(!lookup($2)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int currscope=stack[index1-1]; |
|  |

|  |
| --- |
| int previous\_scope=returnscope($2,currscope); |
|  |

|  |
| --- |
| if(currscope==previous\_scope) |
|  |

|  |
| --- |
| printf("\nError : Redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_dup($2,$1,currscope,nesting()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int scope=stack[index1-1]; |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| insertscope($2,scope); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| | Type ID {push($2);} '=' {strcpy(st1[++top],"=");} E {icg\_1();} ';' |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if( (!(strspn($6,"0123456789")==strlen($6))) && $1==258 && (roo==0)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nError : Type not matching : Line %d\n",printline()); |
|  |

|  |
| --- |
| roo=1; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| if(!lookup($2)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int currscope=stack[index1-1]; |
|  |

|  |
| --- |
| int previous\_scope=returnscope($2,currscope); |
|  |

|  |
| --- |
| if(currscope==previous\_scope) |
|  |

|  |
| --- |
| printf("\nError : Redeclaration of %s : Line %d\n",$2,printline()); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| insert\_dup($2,$1,currscope,nesting()); |
|  |

|  |
| --- |
| int currscope=stack[index1-1]; |
|  |

|  |
| --- |
| int scope=returnscope($2,currscope); |
|  |

|  |
| --- |
| if((scope<=currscope && end[scope]==0) && !(scope==0)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| check\_scope\_update($2,$6,currscope); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int scope=stack[index1-1]; |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| insertscope($2,scope); |
|  |

|  |
| --- |
| check\_scope\_update($2,$6,stack[index1-1]); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| | assignment1 ';' { |
|  |

|  |
| --- |
| if(!lookup($1)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int currscope=stack[index1-1]; |
|  |

|  |
| --- |
| int scope=returnscope($1,currscope); |
|  |

|  |
| --- |
| if(!(scope<=currscope && end[scope]==0) || scope==0) |
|  |

|  |
| --- |
| printf("\nError : Variable %s out of scope : Line %d\n",$1,printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| printf("\nError : Undeclared Variable %s : Line %d\n",$1,printline()); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| | Type ID '[' NUM ']' ';' { |
|  |

|  |
| --- |
| insert($2,ARRAY,nesting()); |
|  |

|  |
| --- |
| int scope=stack[index1-1]; |
|  |

|  |
| --- |
| insertscope($2,scope); |
|  |

|  |
| --- |
| insert($2,$1,nesting()); |
|  |

|  |
| --- |
| storevalue($2,$4,stack[index1-1]); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| | error |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| %% |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| #include "lex.yy.c" |
|  |

|  |
| --- |
| #include<ctype.h> |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int main(int argc, char \*argv[]) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| yyin =fopen(argv[1],"r"); |
|  |

|  |
| --- |
| printf("goto M\n"); |
|  |

|  |
| --- |
| yyparse(); |
|  |

|  |
| --- |
| if(!yyparse()) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("Parsing done\n"); |
|  |

|  |
| --- |
| print(); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("Error\n"); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| fclose(yyin); |
|  |

|  |
| --- |
| return 0; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void yyerror(char \*s) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| printf("\nLine %d : %s %s\n",yylineno,s,yytext); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int printline() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return yylineno; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int nesting() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return count; |
|  |

}

**Symbol Table (symbol.c)**

|  |
| --- |
| #include<stdio.h> |
|  |

|  |
| --- |
| #include<string.h> |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| struct function\_sym |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int function; |
|  |

|  |
| --- |
| int num\_param; |
|  |

|  |
| --- |
| char name[100]; |
|  |

|  |
| --- |
| }func[100]; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| struct sym |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int sno; |
|  |

|  |
| --- |
| char token[100]; |
|  |

|  |
| --- |
| int type[100]; |
|  |

|  |
| --- |
| int tn; |
|  |

|  |
| --- |
| float fvalue; |
|  |

|  |
| --- |
| int nest; |
|  |

|  |
| --- |
| int scope; |
|  |

|  |
| --- |
| }st[100]; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int n=0,arr[10]; |
|  |

|  |
| --- |
| float t[100]; |
|  |

|  |
| --- |
| int iter=0; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void insert\_func(char \*a, int entry, int num\_param) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| strcpy(func[entry].name, a); |
|  |

|  |
| --- |
| func[entry].num\_param = num\_param; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int lookup\_func (char \*a) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int num; |
|  |

|  |
| --- |
| for(num=0; num<function; num++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(strcmp(a, func[num].name)==0) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return num; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return -1; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int number\_param (int a) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return func[a].num\_param; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int returntype\_func(int ct) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return arr[ct-1]; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void insert(char \*name, int type, int count) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| if(lookup(name)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| strcpy(st[n].token,name); |
|  |

|  |
| --- |
| st[n].tn=1; |
|  |

|  |
| --- |
| st[n].type[st[n].tn-1]=type; |
|  |

|  |
| --- |
| st[n].sno=n+1; |
|  |

|  |
| --- |
| st[n].nest=count; |
|  |

|  |
| --- |
| n++; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| for(i=0;i<n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(name,st[i].token)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| st[i].tn++; |
|  |

|  |
| --- |
| st[i].type[st[i].tn-1]=type; |
|  |

|  |
| --- |
| break; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void insert\_dup(char \*name, int type,int s\_c,int count) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| strcpy(st[n].token,name); |
|  |

|  |
| --- |
| st[n].tn=1; |
|  |

|  |
| --- |
| st[n].type[st[n].tn-1]=type; |
|  |

|  |
| --- |
| st[n].sno=n+1; |
|  |

|  |
| --- |
| st[n].scope=s\_c; |
|  |

|  |
| --- |
| st[n].nest=count; |
|  |

|  |
| --- |
| n++; |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void storereturn( int ct, int returntype ) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| arr[ct] = returntype; |
|  |

|  |
| --- |
| return; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void insertscope(char \*a,int s) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| for(i=0;i<n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(a,st[i].token)) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| st[i].scope=s; |
|  |

|  |
| --- |
| break; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int returnscope(char \*a,int cs) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| int max = 0; |
|  |

|  |
| --- |
| for(i=0;i<=n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!(strcmp(a,st[i].token)) && cs>=st[i].scope) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(st[i].scope>=max) |
|  |

|  |
| --- |
| max = st[i].scope; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return max; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int lookup(char \*a) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| for(i=0;i<n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if( !strcmp( a, st[i].token) ) |
|  |

|  |
| --- |
| return 0; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return 1; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int returntype(char \*a,int sct) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| for(i=0;i<=n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(a,st[i].token) && st[i].scope==sct) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return st[i].type[0]; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void check\_scope\_update(char \*a,char \*b,int sc) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i,j,k; |
|  |

|  |
| --- |
| int max=0; |
|  |

|  |
| --- |
| for(i=0;i<=n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(a,st[i].token) && sc>=st[i].scope) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(st[i].scope>=max) |
|  |

|  |
| --- |
| max=st[i].scope; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| for(i=0;i<=n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(a,st[i].token) && max==st[i].scope) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| float temp=atof(b); |
|  |

|  |
| --- |
| for(k=0;k<st[i].tn;k++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(st[i].type[k]==258) |
|  |

|  |
| --- |
| st[i].fvalue=(int)temp; |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| st[i].fvalue=temp; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void storevalue(char \*a,char \*b,int s\_c) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i; |
|  |

|  |
| --- |
| for(i=0;i<=n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(!strcmp(a,st[i].token) && s\_c==st[i].scope) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| st[i].fvalue=atof(b); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| void print() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int i,j; |
|  |

|  |
| --- |
| printf("\nSymbol Table\n\n"); |
|  |

|  |
| --- |
| printf("\nSNo.\tToken\tValue\tScope\tNesting\tType\n"); |
|  |

|  |
| --- |
| for(i=0;i<n;i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(st[i].type[0]==258) |
|  |

|  |
| --- |
| printf("%d\t%s\t%d\t%d\t%d",st[i].sno,st[i].token,(int)st[i].fvalue,st[i].scope,st[i].nest); |
|  |

|  |
| --- |
| else |
|  |

|  |
| --- |
| printf("%d\t%s\t%.1f\t%d\t%d",st[i].sno,st[i].token,st[i].fvalue,st[i].scope,st[i].nest); |
|  |

|  |
| --- |
| for(j=0;j<st[i].tn;j++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| if(st[i].type[j]==258) |
|  |

|  |
| --- |
| printf("\tINT"); |
|  |

|  |
| --- |
| else if(st[i].type[j]==259) |
|  |

|  |
| --- |
| printf("\tFLOAT"); |
|  |

|  |
| --- |
| else if(st[i].type[j]==276) |
|  |

|  |
| --- |
| printf("\tFUNCTION"); |
|  |

|  |
| --- |
| else if(st[i].type[j]==278) |
|  |

|  |
| --- |
| printf("\tARRAY"); |
|  |

|  |
| --- |
| else if(st[i].type[j]==260) |
|  |

|  |
| --- |
| printf("\tVOID"); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| printf("\n"); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| return; |
|  |

}

**Implementation Explanation**

**Files:**

1. **scanner.l:** Lex file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.

2. **parser.y**: Yacc file which has productions for all the loops, conditional statements and expressions. This file also has semantic rules and intermediate code generation rules.

3. **symbol.c**: It is the C file which generates the symbol table. It is included in the yacc file.

4. **test1.c:** The input C code which will be parsed and checked for semantic correctness by executing the lex and yacc files.

**Commands for Execution:**

lex scanner.l

yacc parser.y

gcc y.tab.c -ll -ly -w

./a.out test1.c

Parser checks the C code for syntactical errors and semantic errors and specifies them if present. Otherwise, it displays ‘Parsing Complete’.

In main() of the parser file, the first statement to be printed is: “goto M”. **This ensures that whatever be the order of functions in the input file, the first function to be executed is main.**

**Function Definition and Function Call:**

Every function definition is preceded by a label. For example, a function, sum() is preceded by the label F0. Whenever sum() is called from main(), the statement “goto F0” will be printed. Thus the statements present in the definition of function, sum() will be executed.

After all the statements of sum() have been executed, control of program should return to the statement immediately following the function call in main(). This is done by printing the statements:

goto F0

M0:

After intermediate code has been generated for all statements in the function definition of sum(), the following statement is printed:

goto M0

This works if a function has been called only once in main().

***Multiple calls for the same function***

In this case, we use the label M00 instead of M0, where the first 0 indicates the function number in the function table and the second 0 indicates the fact that this is the first time the function, sum() has been called. The following statement remains the same: “goto M0”.

After 3AC has been generated for all statements in the input file, we add a label M0 and use a temp variable in the following way:

M0:

t=temp==1

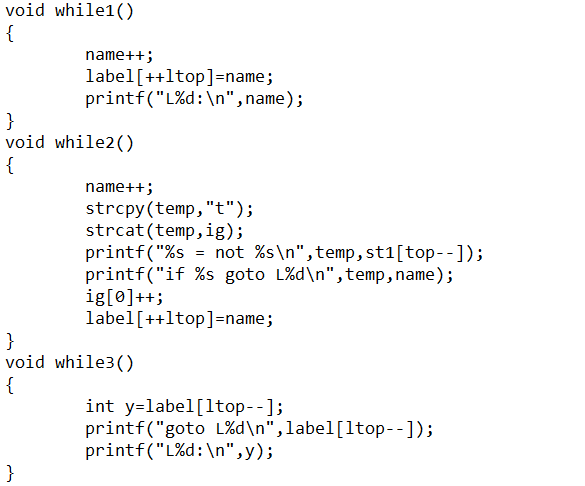
If t go to M0t

Where temp is used to store how many times a particular function has been called. The value of the variable, temp is incremented every time a function is called.

The functions to generate 3AC and the production rules for the C Constructs are as follows:

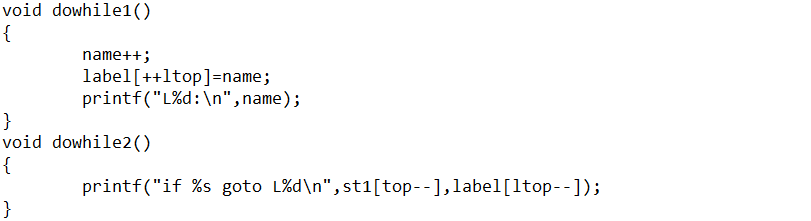
**While loop**





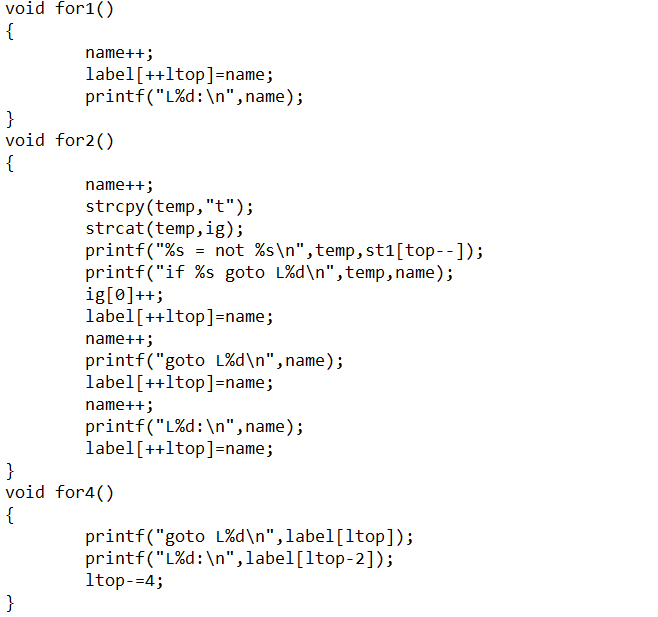
**Do while loop**

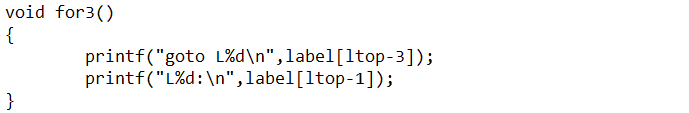




**For loop**

****

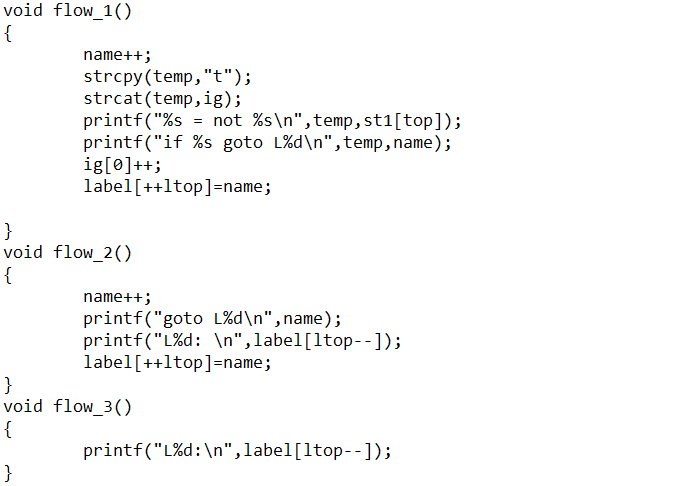
****

****

**If Else**

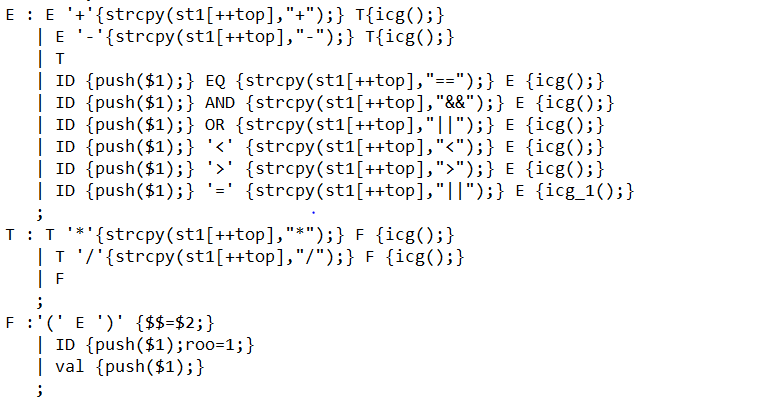






**Expressions**

The precedence of the operators has been mentioned at the start of the code. This ensures proper evaluation of expressions.

****

**Back-patching**

There is a problem of generating label for statements like:

if ( B) then S

When B is false, statement following S needs to be executed. In one-pass translation, B must be translated before S is examined.

Without back-patching, this is done by passing labels as inherited attributes to where the relevant jump instructions were generated, but a separate pass is then needed to bind labels to addresses.

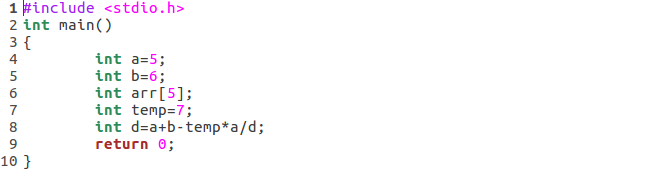
Back-patching can be used to generate code for Boolean expressions and flow-of-control statements in one pass. For this, synthesized attributes *truelist* and *falselist* of non-terminal B are used to manage labels in jumping code for Boolean expressions. In particular, *B.truelist* will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true. *B.falselist* likewise is the list of instructions that eventually get the label to which control goes when B is false. As code is generated for B, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by *B.truelist* and *B.falselist*, as appropriate.

**We have used back-patching while handling multiple calls to same function.**

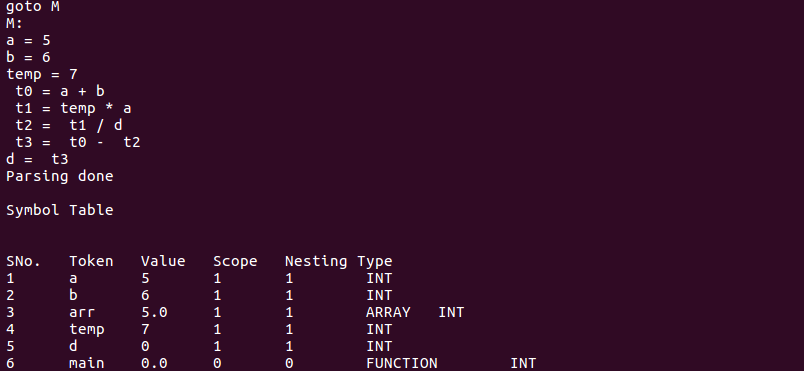
**Test Cases**

**Test Case #1**

**Input**

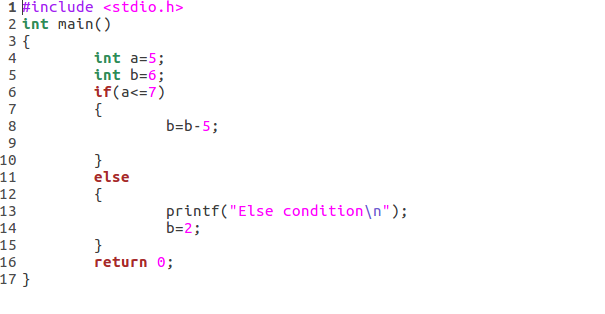


**Output**

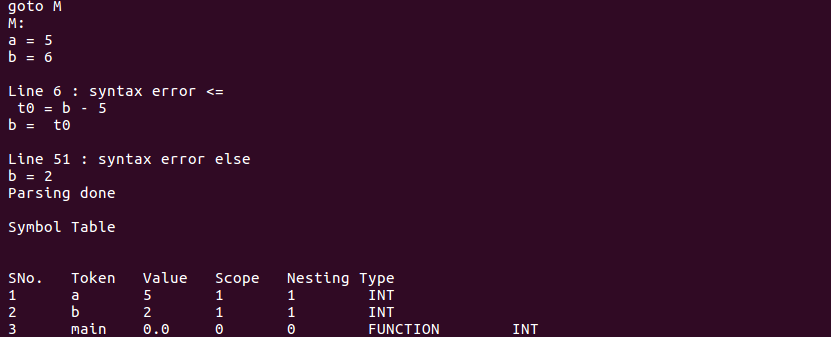


**Test Case #2**

**Input**

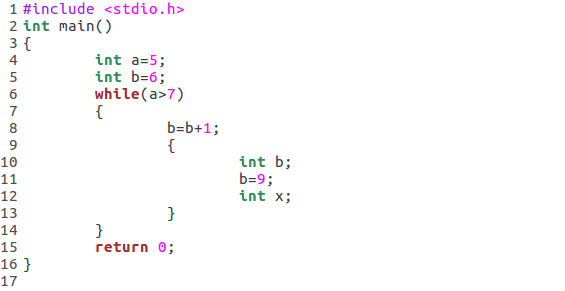
****

**Output**

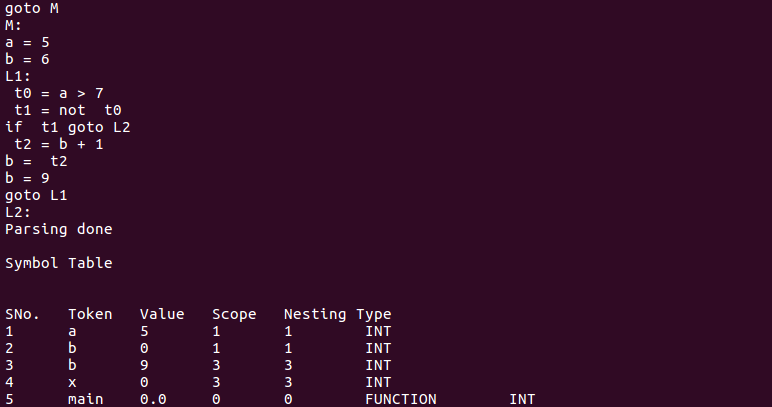
****

**Test Case #3**

**Input**

****

**Output**

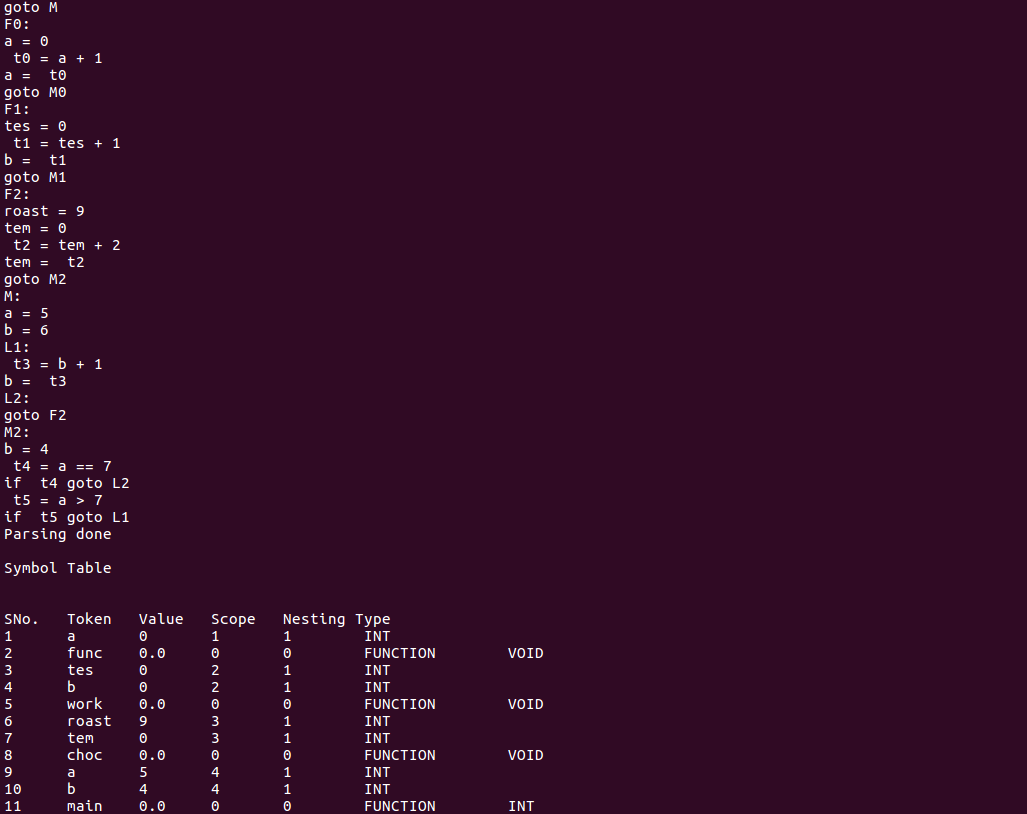
****

**Test Case #4**

**Input**

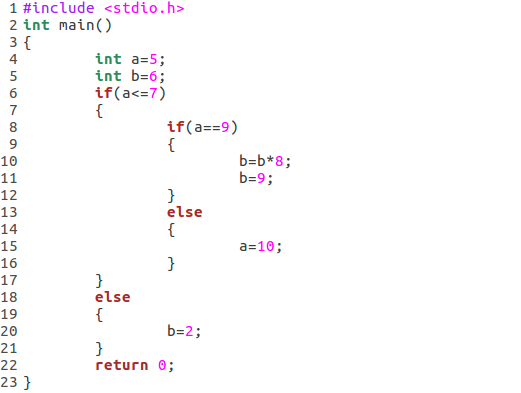


**Output**

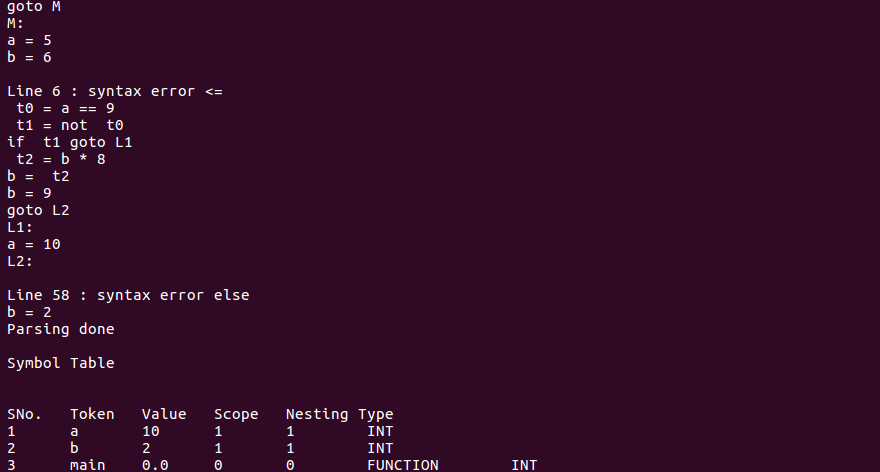
****

**Test Case #5**

**Input**

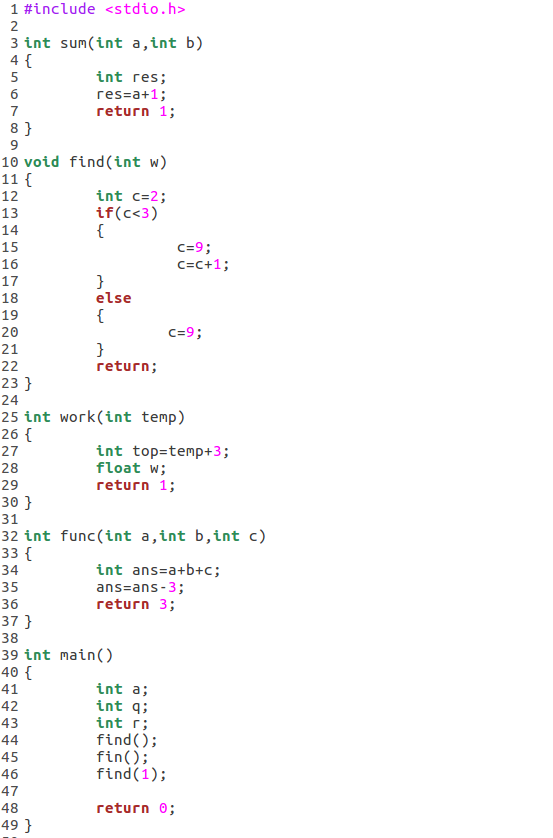
****

**Output**

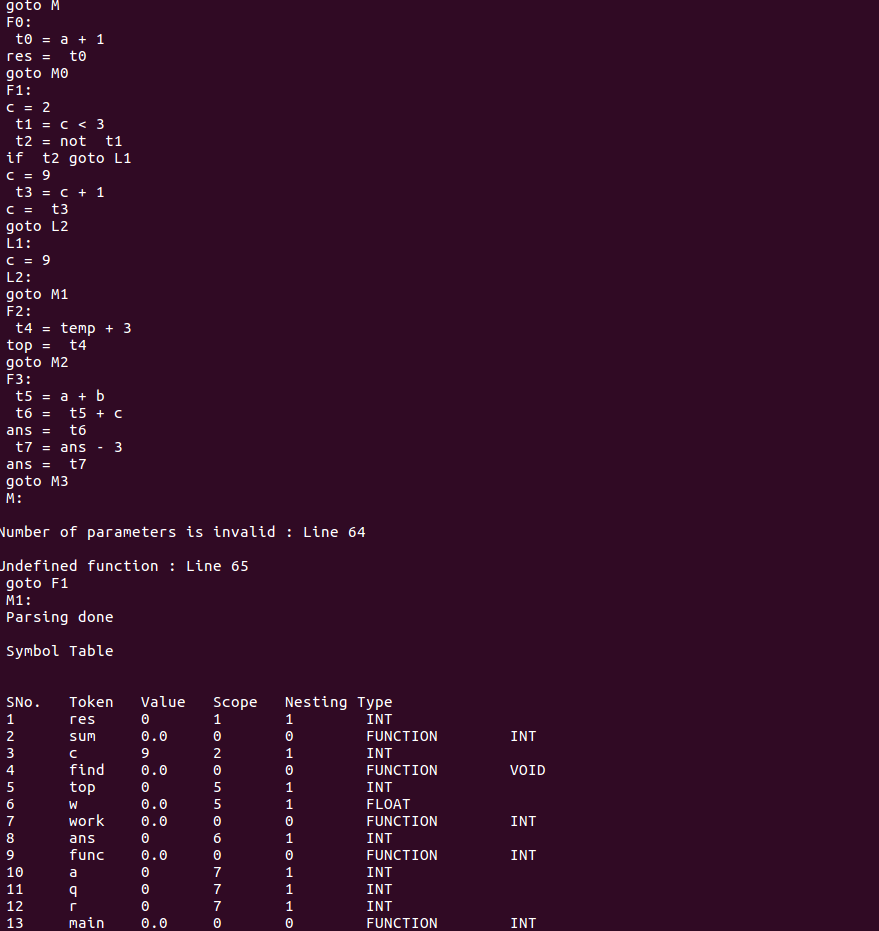
****

**Test Case #6**

**Input**

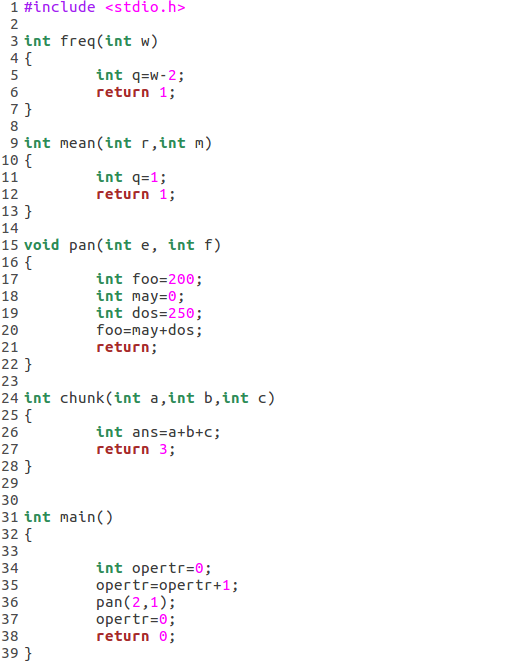
****

**Output**

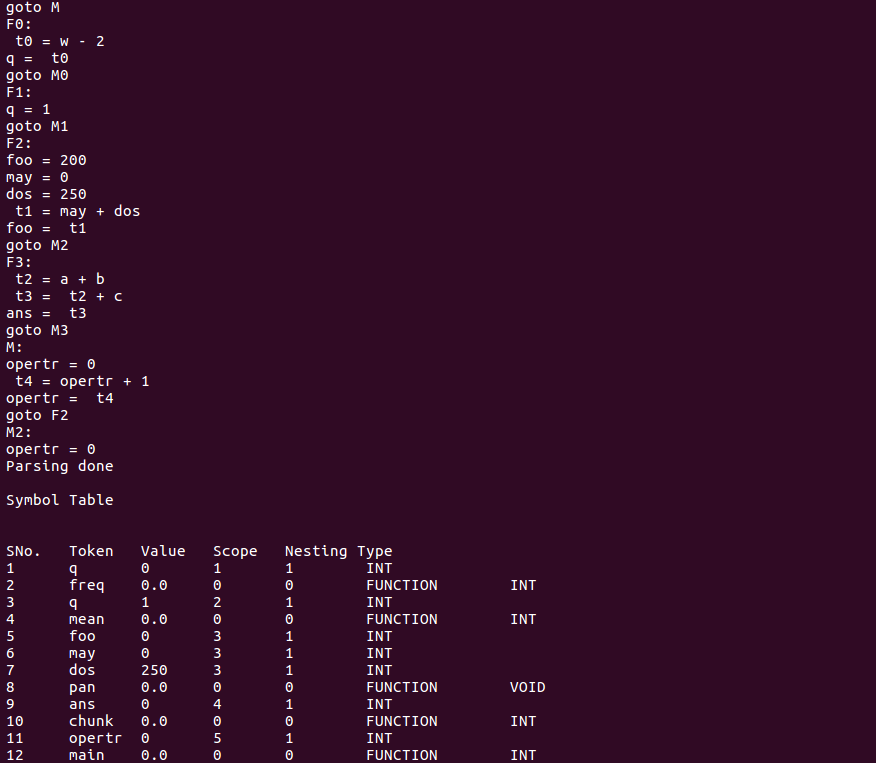


**Test Case # 7**

**Input**



**Output:**

****

**Conclusion and Future work**

Intermediate Code Generation generates 3AC for:

Expressions

Assignments

If Else

While

For

Do while

Functions

We plan to extend ICG to generate 3AC for arrays and pointers.

**References**

* Compilers – Principles, Techniques and Tools - By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
* <https://web.cs.wpi.edu/~kal>
* https://www.tutorialspoint.com