COMPILER ASSIGNMENT 4

GROUP INFORMATION:
150101069: SHIVRAM N GOWTHAM
150101088: PRIYANKAR JAIN
150101085: SHASHANK ANIL HUDDEDAR
150101067: SHASHANK GAREWAL

Yacc/Bison Code:

```
%{
void yyerror(char *s);
extern int mylineno;
extern int yylineno;
extern char *yytext;
#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
#include <string.h>
#include <assert.h>
FILE* fout;
FILE* asmout;
int cur_scope = 0;
int cur_off = 0;
int not_label = 0;
int text_printed = 0;
int tot_func = 0;
int label_cnt = 0;

int is_for[100];
int loop_no[100];
int loop_cnt = 0;

%}

%error-verbose
%start st

%token  EOI
%token  SEMI
%token  MAIN
%token  PLUS
%token  TIMES
%token  LP
%token  RP
%token  MINUS
%token  DIV
%token  NEQ
%token  LE
```

```
%token  LEQ
%token  GR
%token  GRQ
%token  EQ
%token  IF
%token  OPEN_CURL
%token  CLOSE_CURL
%token  ELSE
%token  WHILE
%token  DO
%token  EEQ
%token  SWITCH
%token  CASE
%token  DEFAULT
%token  FOR
%token  BREAK
%token  CONTINUE
%token  LSQBRAC
%token  RSQBRAC
%token  DOT
%token  COLON
%token  COMMA
%token  PLUSPLUS
%token  MINUSMINUS
%token  RETURN
%token  NOT
%token  ID
%token  INT
%token  FLOAT
%token  STRING
%token  AMP
%token  INT_TYPE
%token  FLOAT_TYPE
%token  CHAR_TYPE
%token  CHAR
%token  AND_AND
%token  OR_OR
%token PRINT
%token ENDL

%code requires{

#define NUM_REG 18

enum ele_type {INT_T,FLOAT_T,CHAR_T,BOOL,ERROR};
enum op_type
{_NOT,_PLUS,_MINUS,_AND_AND,_OR_OR,_EEQ,_NEQ,_LEQ,_GRQ,_LE,_GR,_TIMES,_D
IV};


struct func_node{
   char name[50];
```

```
    enum ele_type et;
    int ptr_depth;

    int valid;

    struct node *param_list;
    int num_params;

    struct func_node *next;
};
struct node
{
    char name[50];

    enum ele_type et;
    int ptr_depth;

    int size;
    int* dim_list;
    int st_off;
    int typeSize;
    int prod;
    char* reg_off;
    int valid_off;


    int valid;          /* whether this entry is deleted or not*/
    int scope;

    int expr_val; //init to -1000000


    int num_func;

    char* reg_name;

    struct node *next;
};
struct list_attr{
    struct node* list_var;
    int size;
};
struct list_brac{
        int size;
        int* dim_list;
        char** reg_name;
};



char  Names[NUM_REG][4];
```

```
int used[NUM_REG];


struct func_node *func_table;
struct node cur_func;
struct node *symb_table;

void insert_symb_table(struct node* var);
void add_params_to_sym(struct list_attr lis);
void check_distinct(struct node *attrs, int size);
struct func_node* get_func_entry(char *name);
void insert_func_entry(struct node type,struct node id,struct list_attr attrs);
struct node check_func_exists(char *name);
int match_node_list(struct node *list_var,int lsize,struct node *param_list,int psize,int isProto);
int check_func_proto(struct node type,struct node id,struct list_attr attrs);
int check_param_match(struct node fn,struct list_attr attrs);
int match_node(struct node *n1,struct node *n2);
int exists_in_scope(struct node var);
void patch_type(struct node type,struct list_attr lis);
struct node get_sym(char* var_name,struct list_brac* lsb);
void delete_entries();
void delete_entries_on_ret();
struct node print_comp_error(struct node exp1, struct node exp2);
struct node compatible(struct node exp1,struct node exp2);
struct node print_opt_error(struct node exp1,enum op_type op, struct node exp2);
struct node comp_binopt_type(struct node exp1,enum op_type op, struct node exp2);
struct node comp_opt_type(enum op_type op,struct node exp);
int match_node_exact(struct node n1,struct node n2);
int add(int a,int b);
int mul(int a,int b);
int sub(int a,int b);
int _div(int a,int b);
void freename(char *s);
char *newname();

}

%union{
   struct list_brac lb;


   int list_num;
   int label_num;
   struct node n1;

   struct list_attr lis_attr;
}

%type<list_num> ptr
%type<lb> br_list arr_index br_list_dyn arr_index_dyn
%type<n1> id ide_id_decl type fis ide_id ide_id_lhs function_call expression ide_id_amp
%type<n1> expression_and term factor arithmetic multdiv final
```

```
%type<lis_attr> decl_list list_params list_call_params
%type<label_num> ifexp while_exp for_exp mif iftrue while_stmnt while_head for_stmnt for_cond
for_head

%%

st      : decl SEMI st | proto;
epsilon : ;

decl : type decl_list {
   patch_type($1,$2);
};



type : INT_TYPE { $$.et = INT_T; $$.ptr_depth=0; $$.size = 0;}
             | FLOAT_TYPE { $$.et = FLOAT_T; $$.ptr_depth=0;  $$.size = 0;}
             | CHAR_TYPE { $$.et = CHAR_T; $$.ptr_depth=0;  $$.size = 0;}
             | INT_TYPE ptr { $$.et = INT_T; $$.ptr_depth=$2;  $$.size = 0;}
             | FLOAT_TYPE ptr { $$.et = FLOAT_T; $$.ptr_depth=$2;  $$.size = 0;}
             | CHAR_TYPE ptr { $$.et = CHAR_T; $$.ptr_depth=$2;  $$.size = 0;}
             ;
ptr : TIMES {$$ = 1;}| TIMES ptr {$$ = $2 + 1;};

decl_list : ide_id_decl {
             $$.list_var = (struct node*)malloc(sizeof(struct node));
             $$.list_var[0] = $1; $$.size = 1;}
          | ide_id_decl COMMA decl_list {
             $$.size = 1 + $3.size;
             $$.list_var = (struct node*)malloc(sizeof(struct node)*$$.size);

             int iter = 0;
             for(iter = 0;iter < $3.size; ++iter) $$.list_var[iter] = $3.list_var[iter];
             $$.list_var[$3.size] = $1;
           }
          ;

id : ID {strcpy($$.name,strdup(yytext));};


ide_id_decl : id {
       strcpy($$.name,$1.name);
       $$.size = 0;
       $$.dim_list = NULL;
   }
   | id  br_list{
       strcpy($$.name,$1.name);
       $$.size = $2.size;
       $$.dim_list = $2.dim_list;
   };
```

```
arr_index : LSQBRAC expression RSQBRAC {
   if($2.et == INT_T && $2.ptr_depth == 0 && $2.size == 0 && $2.expr_val > 0){
      $$.size = 1;
      $$.dim_list = (int*)malloc(sizeof(int));
      $$.dim_list[0] = $2.expr_val;
   }
   else{
      printf("index must be int and +ve constant\n");
      $$.size = 1;
      $$.dim_list = (int*)malloc(sizeof(int));
      $$.dim_list[0] = 1;
   }
   freename($2.reg_name);
   freename($2.reg_off);
};
br_list : arr_index {
          $$ = $1;
       }
       | arr_index br_list {
          $$.dim_list = (int*)malloc(sizeof(int)*($2.size+1));

          int iter = 0;
          for(iter = 0;iter < $2.size; ++iter) $$.dim_list[iter] = $2.dim_list[iter];

          $$.size = $2.size + 1;
          $$.dim_list[$2.size] = $1.dim_list[0];

   }
   ;




proto_head : type id LP list_params RP {insert_func_entry($1,$2,$4);}
proto : proto_head SEMI proto
       | st2
       ;

func_head :  type id LP list_params RP {
          if(check_func_proto($1,$2,$4) == 0){
             printf("Prototype mismatch/not declared\n");
          }
          cur_scope++;

          if(!text_printed){
             text_printed = 1;
             fprintf(asmout, ".text\n.globl main\n" );
          }
          fprintf(asmout, "%s: \n",$2.name );

          int iter = 0;
```

```
            for(;iter < $4.size;++iter){
               $4.list_var[iter].st_off = cur_off;
               // printf("parameter name is %s for func %s\n",$4.list_var[iter].name,$2.name );
               cur_off += $4.list_var[iter].prod*$4.list_var[iter].typeSize;
            }

            add_params_to_sym($4);
            cur_func = $1;
            cur_func.num_func = ++tot_func;
            }
        ;
int_head: INT_TYPE MAIN LP RP{
        cur_func.et = INT_T;cur_func.size = 0;cur_func.ptr_depth = 0;
        cur_scope++;
        cur_func.num_func = ++tot_func;

        if(!text_printed){
           text_printed = 1;
           fprintf(asmout, ".text\n" );
        }
        fprintf(asmout, "main: \n" );
        fprintf(asmout, "move $fp,$sp\n" );
    };

st2     : int_head OPEN_CURL s CLOSE_CURL {
                        delete_entries();
                        cur_scope--;

                           assert(cur_scope == 0 && cur_off == 0);

                           fprintf(asmout, "li $v0,10\nsyscall\n");
                        }

        | func_head func_body st2
        ;
func_body: OPEN_CURL s CLOSE_CURL{
    delete_entries();
    cur_scope--;
    assert(cur_scope == 0 && cur_off == 0);
    fprintf(asmout, "li $v1,0\n");
    fprintf(asmout, "jr $ra\n");
};




list_params     : type ide_id_decl {
                $$.list_var = (struct node*)malloc(sizeof(struct node));
                $2.et = $1.et;
                $2.ptr_depth = $1.ptr_depth;
```

```
            if($2.size){
               $2.prod = 1;
               int i = 0;
               for(;i < $2.size;++i) $2.prod *= $2.dim_list[i];
            }
            else $2.prod = 1;

            int cprod = $2.prod;

            if(1){
               $2.typeSize = 4; /// MIPS pointer size
            }
            $$.list_var[0] = $2;
            $$.size = 1;

         }
         | type ide_id_decl COMMA list_params
         {
            $$.size = 1 + $4.size;
            $$.list_var = (struct node*)malloc(sizeof(struct node)*$$.size);

            int iter = 0;
            for(iter = 0;iter < $4.size; ++iter) $$.list_var[iter] = $4.list_var[iter];

            $2.et = $1.et;
            $2.ptr_depth = $1.ptr_depth;

            if($2.size){
               $2.prod = 1;
               int i = 0;
               for(;i < $2.size;++i) $2.prod *= $2.dim_list[i];
            }
            else $2.prod = 1;

            int cprod = $2.prod;

            if(1){
               $2.typeSize = 4; /// MIPS pointer size
            }

            $$.list_var[$4.size] = $2;
         }
         ;

fis    : INT   {
            $$.et = INT_T; $$.ptr_depth = 0;
            $$.size = 0;
            $$.dim_list = NULL;
            $$.expr_val = atoi(strdup(yytext));

            $$.reg_name = newname();
            fprintf(asmout, "li %s,%d\n",$$.reg_name,$$.expr_val );
```

```
        }
        | FLOAT  {$$.et = FLOAT_T; $$.ptr_depth = 0; $$.size = 0;$$.dim_list = NULL; $$.expr_val
= -1000000;}
        | CHAR   {
            $$.et = CHAR_T; $$.ptr_depth = 0; $$.size = 0;$$.dim_list = NULL; $$.expr_val =
yytext[1];

            $$.reg_name = newname();
            fprintf(asmout, "li %s,%d\n",$$.reg_name,$$.expr_val );

                //printf("CHAR VALUE IS %d %s\n",$$.expr_val,yytext);
        }
        | MINUS INT {$$.et = INT_T; $$.ptr_depth = 0; $$.size = 0; $$.dim_list = NULL; $
$.expr_val = -atoi(strdup(yytext));

                $$.reg_name = newname();
                fprintf(asmout, "li %s,%d\n",$$.reg_name,$$.expr_val );
            }
        ;

arr_index_dyn : LSQBRAC expression RSQBRAC {
    if(!($2.et == INT_T && $2.ptr_depth == 0 && $2.size == 0)){
        printf("index must be int type\n");
    }
    $$.size = 1;
    $$.reg_name = (char**)malloc(sizeof(char*)*(1));
    $$.reg_name[0] = (char*)malloc(4);
    strcpy($$.reg_name[0],$2.reg_name);
};
br_list_dyn : arr_index_dyn {
            $$ = $1;
        }
        | arr_index_dyn br_list_dyn {
            $$.reg_name = (char**)malloc(sizeof(char*)*($2.size+1));

            int iter = 0;
            for(iter = 0;iter < $2.size; ++iter) {
                $$.reg_name[iter] = (char*)malloc(4);
                strcpy($$.reg_name[iter],$2.reg_name[iter]);
            }

            $$.size = $2.size + 1;
            $$.reg_name[$2.size] = (char*)malloc(4);
            strcpy($$.reg_name[$2.size],$1.reg_name[0]);
    }
    ;



ide_id : id {
    ///global vars sepeate access
```

```
    ///dynamic access a[i][j]


        $$ = get_sym($1.name,NULL);
        $$.expr_val = -1000000;
        if($$.size == 0){
           char* reg_name = (char*)malloc(sizeof(char)*4);
           strcpy(reg_name,"$a0");
           fprintf(asmout, "addi %s,$fp,-%d\n",reg_name,$$.st_off + $$.typeSize );
           fprintf(asmout, "sub %s,%s,%s\n",reg_name,reg_name,$$.reg_off );
           freename($$.reg_off);
           $$.valid_off = 0;

           $$.reg_name = newname();
           if($$.typeSize == 4) fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );
           else fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );

        }

        } | id br_list_dyn {
           $$ = get_sym($1.name,&($2));
           $$.expr_val = -1000000;
           if($$.size == 0){
              char* reg_name = (char*)malloc(sizeof(char)*4);
              strcpy(reg_name,"$a0");
              fprintf(asmout, "addi %s,$fp,-%d\n",reg_name,$$.st_off + $$.typeSize );
              fprintf(asmout, "sub %s,%s,%s\n",reg_name,reg_name,$$.reg_off );
              freename($$.reg_off);
              $$.valid_off = 0;

              $$.reg_name = newname();
              if($$.typeSize == 4) fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );
              else fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );

           }
        };


ide_id_amp : id {
    ///global vars sepeate access


    ///dynamic access a[i][j]


        $$ = get_sym($1.name,NULL);
        $$.expr_val = -1000000;


        } | id br_list_dyn {
```

```
            $$ = get_sym($1.name,&($2));
            $$.expr_val = -1000000;

        };


ide_id_lhs : id {

        $$ = get_sym($1.name,NULL);
        $$.expr_val = -1000000;
        // printf("reached here \n");
        if($$.size == 0){
            fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
            fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off );
            //freename($$.reg_off);
            //$$.valid_off = 0;

            $$.reg_name = newname();
            if($$.typeSize == 4) fprintf(asmout, "lw %s,($a0)\n",$$.reg_name );
            else fprintf(asmout, "lw %s,($a0)\n",$$.reg_name );
            // printf("reached here 2\n");
        }
        else{
            printf("id cant be array in this case\n");
        }

        } | id br_list_dyn {
            $$ = get_sym($1.name,&($2));
            $$.expr_val = -1000000;
            if($$.size == 0){
                char* reg_name = (char*)malloc(sizeof(char)*4);
                strcpy(reg_name,"$a0");
                fprintf(asmout, "addi %s,$fp,-%d\n",reg_name,$$.st_off + $$.typeSize );
                fprintf(asmout, "sub %s,%s,%s\n",reg_name,reg_name,$$.reg_off );
                //freename($$.reg_off);
                //$$.valid_off = 0;

                $$.reg_name = newname();
                if($$.typeSize == 4) fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );
                else fprintf(asmout, "lw %s,(%s)\n",$$.reg_name,reg_name );

            }
            else{
                printf("id cant be array in this case\n");
            }
        };

list_call_params        : expression    {
                            $$.list_var = (struct node*)malloc(sizeof(struct node));
                            $$.list_var[0] = $1;
                            $$.size = 1;
                        }
```

```
                    | expression COMMA list_call_params {
                       $$.size = 1 + $3.size;

                       $$.list_var = (struct node*)malloc(sizeof(struct node)*$$.size);

                       int iter = 0;
                       for(iter = 0;iter < $3.size; ++iter) $$.list_var[iter] = $3.list_var[iter];
                       $$.list_var[$3.size] = $1;
                    }
                    ;

s      : epsilon {fprintf(fout,"s -> epsilon\n");}
       | other_than_if s {fprintf(fout,"s -> uif s\n");}
       | mif s {fprintf(fout,"s -> mif s\n");}
       ;

function_call   : id LP list_call_params RP   {
   $$ = check_func_exists($1.name);
   if($$.valid == 1){
       int i=0;
       for(;i<NUM_REG;++i){
          fprintf(asmout, "addi $sp,$sp,-4\n" );
          fprintf(asmout, "sw %s,($sp)\n",Names[i]);
       }
       fprintf(asmout, "addi $sp,$sp,-4\n" );
       fprintf(asmout, "sw $fp,($sp)\n");
       fprintf(asmout, "addi $sp,$sp,-4\n" );
       fprintf(asmout, "sw $ra,($sp)\n");

       fprintf(asmout, "move $a0,$fp\n" );
       fprintf(asmout, "move $fp,$sp\n" );

       check_param_match($$,$3);

       fprintf(asmout, "jal %s\n",$1.name );

       fprintf(asmout, "lw $ra,($sp)\n");
       fprintf(asmout, "addi $sp,$sp,4\n" );
       fprintf(asmout, "lw $fp,($sp)\n");
       fprintf(asmout, "addi $sp,$sp,4\n" );

       for(i=NUM_REG-1;i>=0;i--){
          fprintf(asmout, "lw %s,($sp)\n",Names[i]);
          fprintf(asmout, "addi $sp,$sp,4\n" );
       }

   }
};

for_head : FOR LP ide_id_lhs EQ expression SEMI {
   if($5.size){
      fprintf(asmout, "addi $a0,$fp,-%d\n",$5.st_off + $5.typeSize );
```

```
        fprintf(asmout, "sub $a0,$a0,%s\n",$5.reg_off);
        freename($5.reg_off);
        $5.valid_off = 0;

        $5.reg_name = newname();
        $5.ptr_depth += $5.size;
        $5.size = 0;
        fprintf(asmout, "move %s,$a0\n",$5.reg_name );
        $5.expr_val = -1000000;
    }
    compatible($3,$5);

    fprintf(asmout, "addi $a0,$fp,-%d\n",$3.st_off + $3.typeSize);
    fprintf(asmout, "sub $a0,$a0,%s\n",$3.reg_off );

    if($3.typeSize == 1) {
        fprintf(asmout, "sw %s,($a0)\n",$5.reg_name );
    }
    else {
        fprintf(asmout, "sw %s,($a0)\n",$5.reg_name );
    }

    freename($3.reg_off);freename($5.reg_off);
    freename($3.reg_name);freename($5.reg_name);

    $$ = ++label_cnt;

    loop_cnt++;
    is_for[loop_cnt] = 1;
    loop_no[loop_cnt] = $$;

    fprintf(asmout, "START_LABEL%d:\n",$$ );
};
for_cond: for_head expression SEMI{
    $$ = $1;

    if( $2.size > 0 || $2.ptr_depth >0)
        printf("Boolean required in expression\n");
    else{
        fprintf(asmout, "beq %s,$zero,LABEL%d\n",$2.reg_name,$$ );

    }
    freename($2.reg_name);
    freename($2.reg_off);
    fprintf(asmout, "j FOR_START%d\n",$$ );
    fprintf(asmout, "FOR_INC%d:\n",$$ );
};
for_exp : for_cond expression RP
            {
                $$=$1;
                fprintf(asmout, "j START_LABEL%d\n",$$ );
                fprintf(asmout, "FOR_START%d:\n",$$ );
```

```
            };

for_stmnt  :  for_exp opc s CLOSE_CURL {
   $$ = $1;
   fprintf(asmout, "j FOR_INC%d\n",$$ );
   fprintf(asmout, "LABEL%d:\n",$1 );
    delete_entries(); cur_scope--;

   loop_cnt--;

};

while_head: WHILE LP{
   $$ = ++label_cnt;

   loop_cnt++;
   is_for[loop_cnt] = 0;
   loop_no[loop_cnt] = $$;

   fprintf(asmout, "START_LABEL%d:\n",$$ );

};
while_exp : while_head expression RP
          {
             $$ = $1;
             if($2.size > 0 || $2.ptr_depth >0)
                printf("Boolean required in expression\n");
             else{
                fprintf(asmout, "beq %s,$zero,LABEL%d\n",$2.reg_name,$$ );

             }
             freename($2.reg_name);
             freename($2.reg_off);
          };
while_stmnt  : while_exp opc s CLOSE_CURL  {
   fprintf(asmout, "j START_LABEL%d\n",$1 );
   fprintf(asmout, "LABEL%d:\n",$1 );
    delete_entries(); cur_scope--;

   loop_cnt--;
};


ifexp: IF LP expression RP
       {
          if($3.size > 0 || $3.ptr_depth >0){
             printf("Boolean expected\n");
          }
          else{
             $$ = ++label_cnt;
             fprintf(asmout, "beq %s,$zero,LABEL%d\n",$3.reg_name,$$ );
          }
```

```
            freename($3.reg_name);
            freename($3.reg_off);
        };
iftrue: ifexp opc s CLOSE_CURL{
        fprintf(asmout, "j END_LABEL%d\n",$1 );
        fprintf(asmout, "LABEL%d:\n",$1 );
        $$ = $1;
         delete_entries(); cur_scope--;
};
mif  : iftrue ELSE opc s CLOSE_CURL {
          fprintf(asmout, "END_LABEL%d:\n",$1 );
           delete_entries(); cur_scope--;
        }
        | iftrue {
            fprintf(asmout, "END_LABEL%d:\n",$1 );
        }
        ;


opc: OPEN_CURL {cur_scope++;};

other_than_if      :   ptr ide_id_lhs EQ expression SEMI {
                        if($4.size){
                           fprintf(asmout, "addi $a0,$fp,-%d\n",$4.st_off + $4.typeSize );
                           fprintf(asmout, "sub $a0,$a0,%s\n",$4.reg_off);
                           freename($4.reg_off);
                           $4.valid_off = 0;

                           $4.reg_name = newname();
                           $4.ptr_depth += $4.size;
                           $4.size = 0;
                           fprintf(asmout, "move %s,$a0\n",$4.reg_name );
                           $4.expr_val = -1000000;
                        }
                        if($1 > $2.ptr_depth){
                           printf("Error dereferencing\n");
                        }
                        else{
                           int ctr=0;
                           fprintf(asmout, "move $a1,%s\n",$2.reg_name );
                           for(;ctr < $1-1;++ctr){
                              fprintf(asmout, "lw $a2,($a1)\n" );
                              fprintf(asmout, "move $a1,$a2\n" );
                           }

                           $2.ptr_depth -= $1;
                           compatible($2,$4);

                           fprintf(asmout, "sw %s,($a1)\n",$4.reg_name);
                        }
                        freename($2.reg_name);
```

```
        freename($2.reg_off);
        freename($4.reg_name);
        freename($4.reg_off);
    }


| ide_id_lhs EQ expression SEMI  {
    if($3.size){
        fprintf(asmout, "addi $a0,$fp,-%d\n",$3.st_off + $3.typeSize );
        fprintf(asmout, "sub $a0,$a0,%s\n",$3.reg_off);
        freename($3.reg_off);
        $3.valid_off = 0;

        $3.reg_name = newname();
        $3.ptr_depth += $3.size;
        $3.size = 0;
        fprintf(asmout, "move %s,$a0\n",$3.reg_name );
        $3.expr_val = -1000000;
    }
    compatible($1,$3);

    fprintf(asmout, "addi $a0,$fp,-%d\n",$1.st_off + $1.typeSize);
    fprintf(asmout, "sub $a0,$a0,%s\n",$1.reg_off );

    if($1.typeSize == 1) {
        fprintf(asmout, "sw %s,($a0)\n",$3.reg_name );
    }
    else {
        fprintf(asmout, "sw %s,($a0)\n",$3.reg_name );
    }

    freename($1.reg_off);freename($3.reg_off);
    freename($1.reg_name);freename($3.reg_name);
}


| CONTINUE SEMI  {
    if(!loop_cnt) printf("continue without loop\n");
    if(is_for[loop_cnt]){
        fprintf(asmout, "j FOR_INC%d\n",loop_no[loop_cnt] );
    }
    else{
        fprintf(asmout, "j START_LABEL%d\n",loop_no[loop_cnt]);
    }
}
| BREAK SEMI {
    if(!loop_cnt) printf("break without loop\n");
    fprintf(asmout, "j LABEL%d\n",loop_no[loop_cnt] );
}

| while_stmnt  {fprintf(fout,"other_than_if-> while_stmnt\n");}
| for_stmnt  {fprintf(fout,"other_than_if-> for_stmnt\n");}
```

```
| opc s CLOSE_CURL   { delete_entries(); cur_scope--;}
| decl SEMI
| expression SEMI{
    freename($1.reg_name);
    freename($1.reg_off);
}
| PRINT expression SEMI{
    if($2.size > 0){
        printf("cannot print array expression\n");
    }
    else{
        if($2.et == CHAR_T && $2.ptr_depth == 0) fprintf(asmout, "li $v0,11\n" );
        else fprintf(asmout, "li $v0,1\n" );

        fprintf(asmout,"move $a0,%s\n",$2.reg_name);
        fprintf(asmout, "syscall\n" );
    }
    freename($2.reg_name);
    freename($2.reg_off);
}
| PRINT ENDL SEMI{
    fprintf(asmout, "li $v0,11\n" );
    fprintf(asmout,"li $a0,10\n");
    fprintf(asmout, "syscall\n" );
}
| RETURN expression SEMI  {
    if($2.size){
        fprintf(asmout, "addi $a0,$fp,-%d\n",$2.st_off + $2.typeSize );
        fprintf(asmout, "sub $a0,$a0,%s\n",$2.reg_off);
        freename($2.reg_off);
        $2.valid_off = 0;

        $2.reg_name = newname();
        $2.ptr_depth += $2.size;
        $2.size = 0;
        fprintf(asmout, "move %s,$a0\n",$2.reg_name );
        $2.expr_val = -1000000;
    }

    compatible(cur_func,$2);

    fprintf(asmout, "move $v1,%s\n",$2.reg_name );
    delete_entries_on_ret();
    fprintf(asmout, "jr $ra\n" );
    freename($2.reg_name);
    freename($2.reg_off);
}
;
```

```
expression      : expression_and AND_AND expression  { $$ = comp_binopt_type($1,_AND_AND,
$3);}
            | expression_and OR_OR expression  { $$ = comp_binopt_type($1,_OR_OR,$3);}
            | expression_and {$$ = $1;}
            ;

expression_and      : term EEQ expression_and  { $$ = comp_binopt_type($1,_EEQ,$3);}
            | term NEQ expression_and  { $$ = comp_binopt_type($1,_NEQ,$3);}
            | term {$$ = $1;}
            ;

term          : factor  {$$ = $1;}
            | factor LEQ term  { $$ = comp_binopt_type($1,_LEQ,$3);}
            | factor GRQ term  { $$ = comp_binopt_type($1,_GRQ,$3);}
            | factor GR term  { $$ = comp_binopt_type($1,_GR,$3);}
            | factor LE term { $$ = comp_binopt_type($1,_LE,$3);}
            ;

factor        : arithmetic {$$ = $1;}
            | arithmetic MINUS factor { $$ = comp_binopt_type($1,_MINUS,$3);}
            ;

arithmetic      : multdiv {$$ = $1;}
            | multdiv PLUS arithmetic  {$$ = comp_binopt_type($1,_PLUS,$3);}
            ;

multdiv       : final {$$ = $1;}
            | final TIMES multdiv {$$ = comp_binopt_type($1,_TIMES,$3);}
            | final DIV multdiv  {$$ = comp_binopt_type($1,_DIV,$3);}

final :   ptr ide_id {
          // printf("accessing pointer of %d %d\n",$2.ptr_depth,$2.size);
          if($1 > $2.ptr_depth + $2.size ){
             printf("Error dereferencing\n");
             $$.et = ERROR;
             $$.reg_name = (char*)malloc(1);
             $$.reg_off = (char*)malloc(1);
          }else{
             $$ = $2;
             if($$.size == 0){
                int ctr=0;
                fprintf(asmout, "move $a1,%s\n",$$.reg_name );
                for(;ctr < $1-1;++ctr){
                   fprintf(asmout, "lw $a2,($a1)\n" );
                   fprintf(asmout, "move $a1,$a2\n" );
                }
                int sz = 4;
                if($$.et == CHAR_T && $$.ptr_depth == $1) sz = 4;

                if(sz == 4) fprintf(asmout, "lw $a2,($a1)\n");
                else fprintf(asmout, "lw $a2,($a1)\n");
```

```
            fprintf(asmout, "move %s,$a2\n",$$.reg_name );
            $$.ptr_depth -= $1;
            if($$.ptr_depth == 0 && $$.et == CHAR_T) $$.typeSize = 4;
        }
        else{
            fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
            fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off);
            freename($$.reg_off);
            $$.valid_off = 0;

            $$.reg_name = newname();
            $$.ptr_depth += $$.size;
            $$.size = 0;

            int ctr=0;
            fprintf(asmout, "move $a1,$a0\n" );
            for(;ctr < $1-1;++ctr){
                fprintf(asmout, "lw $a2,($a1)\n" );
                fprintf(asmout, "move $a1,$a2\n" );
            }
            int sz = 4;
            if($$.et == CHAR_T && $$.ptr_depth == $1) sz = 4;

            if(sz == 4) fprintf(asmout, "lw $a2,($a1)\n");
            else fprintf(asmout, "lw $a2,($a1)\n");

            fprintf(asmout, "move %s,$a2\n",$$.reg_name );
            $$.ptr_depth -= $1;
            if($$.ptr_depth == 0 && $$.et == CHAR_T) $$.typeSize = 4;

        }
    }
    $$.expr_val = -1000000;
}
| AMP ide_id_amp {
    $$ = $2;

    fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
    fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off);
    freename($$.reg_off);
    $$.valid_off = 0;

    $$.reg_name = newname();
    $$.ptr_depth += $$.size;
    $$.size = 0;
    fprintf(asmout, "move %s,$a0\n",$$.reg_name );
    $$.expr_val = -1000000;
    $$.ptr_depth++;
}
| ptr LP expression RP {
    if($1 > $3.ptr_depth + $3.size ){
        printf("Error dereferencing\n");
```

```
    $$.et = ERROR;
    $$.reg_name = (char*)malloc(1);
    $$.reg_off = (char*)malloc(1);
  }else{
    $$ = $3;
    if($$.size == 0){
       int ctr=0;
       fprintf(asmout, "move $a1,%s\n",$$.reg_name );
       for(;ctr < $1-1;++ctr){
          fprintf(asmout, "lw $a2,($a1)\n" );
          fprintf(asmout, "move $a1,$a2\n" );
       }
       int sz = 4;
       if($$.et == CHAR_T && $$.ptr_depth == $1) sz = 4;

       if(sz == 4) fprintf(asmout, "lw $a2,($a1)\n");
       else fprintf(asmout, "lw $a2,($a1)\n");

       fprintf(asmout, "move %s,$a2\n",$$.reg_name );
       $$.ptr_depth -= $1;
       if($$.ptr_depth == 0 && $$.et == CHAR_T) $$.typeSize = 4;
    }
    else{
       fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
       fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off);
       freename($$.reg_off);
       $$.valid_off = 0;

       $$.reg_name = newname();
       $$.ptr_depth += $$.size;
       $$.size = 0;

       int ctr=0;
       fprintf(asmout, "move $a1,$a0\n" );
       for(;ctr < $1-1;++ctr){
          fprintf(asmout, "lw $a2,($a1)\n" );
          fprintf(asmout, "move $a1,$a2\n" );
       }
       int sz = 4;
       if($$.et == CHAR_T && $$.ptr_depth == $1) sz = 4;

       if(sz == 4) fprintf(asmout, "lw $a2,($a1)\n");
       else fprintf(asmout, "lw $a2,($a1)\n");

       fprintf(asmout, "move %s,$a2\n",$$.reg_name );
       $$.ptr_depth -= $1;
       if($$.ptr_depth == 0 && $$.et == CHAR_T) $$.typeSize = 4;

    }

  }
```

```
    $$.expr_val = -1000000;
}


| type LP expression RP {
    $$ = $3;
    $1 = compatible($1,$3);
    $$.et = $1.et;
    $$.expr_val = $1.expr_val;
}
| ide_id  {
        $$ = $1;
        $$.expr_val = -1000000;
    }
| PLUSPLUS ide_id_lhs  {
    $$ = comp_opt_type(_PLUS,$2);
    if($$.ptr_depth == 0){
        fprintf(asmout, "addi %s,%s,1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,4\n",$$.reg_name,$$.reg_name );
    }

    fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
    fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off );

    fprintf(asmout, "sw %s,($a0)\n",$$.reg_name);
}
| MINUSMINUS ide_id_lhs  {
   $$ = comp_opt_type(_MINUS,$2);
   if($$.ptr_depth == 0){
        fprintf(asmout, "addi %s,%s,-1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,-4\n",$$.reg_name,$$.reg_name );
    }

    fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
    fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off );

    fprintf(asmout, "sw %s,($a0)\n",$$.reg_name);
}
|ide_id_lhs PLUSPLUS  {
    $$ = comp_opt_type(_PLUS,$1);
    if($$.ptr_depth == 0){
        fprintf(asmout, "addi %s,%s,1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,4\n",$$.reg_name,$$.reg_name );
    }

    fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
```

```
    fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off );

    fprintf(asmout, "sw %s,($a0)\n",$$.reg_name);

    if($$.ptr_depth == 0){
        fprintf(asmout, "addi %s,%s,-1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,-4\n",$$.reg_name,$$.reg_name );
    }
}
| ide_id_lhs MINUSMINUS  {
    $$ = comp_opt_type(_MINUS,$1);
    if($$.ptr_depth == 0 ){
        fprintf(asmout, "addi %s,%s,-1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,-4\n",$$.reg_name,$$.reg_name );
    }

    fprintf(asmout, "addi $a0,$fp,-%d\n",$$.st_off + $$.typeSize );
    fprintf(asmout, "sub $a0,$a0,%s\n",$$.reg_off );

    fprintf(asmout, "sw %s,($a0)\n",$$.reg_name);

    if($$.ptr_depth == 0 ){
        fprintf(asmout, "addi %s,%s,1\n",$$.reg_name,$$.reg_name );
    }
    else{
        fprintf(asmout, "addi %s,%s,4\n",$$.reg_name,$$.reg_name );
    }
}
| NOT LP expression RP  {
    $$ = comp_opt_type(_NOT,$3);
    fprintf(asmout, "bne %s,$zero,NOT_LABEL%d\n",$$.reg_name,not_label );
    fprintf(asmout, "li %s,1\n",$$.reg_name );
    fprintf(asmout, "j NOT_LABEL_END%d\n",not_label );
    fprintf(asmout, "NOT_LABEL%d:\n li %s,0\n",not_label,$$.reg_name );
    fprintf(asmout, "NOT_LABEL_END%d:\n",not_label );
    not_label++;
}
| LP expression RP  {
        $$ = $2;

}
| fis  {
        $$ = $1;
    }
| function_call  {
    $$ = $1;
    $$.reg_name = newname();
    fprintf(asmout, "move %s,$v1\n",$$.reg_name );
```

```
            $$.expr_val = -1000000;
        }
        ;
%%

void insert_symb_table(struct node* var)
{
    if (var == NULL)
        return ;

    var->valid = 1;
    var->next = symb_table;
    symb_table = var;
}

void add_params_to_sym(struct list_attr lis)
{
    int i=0;
    while(i < lis.size)
    {
        struct node* param_list = &lis.list_var[i];
        param_list->scope = 1;

        if(exists_in_scope(*param_list)){
            printf("Parameter already defined %s\n",param_list->name);
        }else{
            // printf("insertiing %s\n",param_list->name);
            insert_symb_table(param_list);
        }

        i++;
    }
}

void check_distinct(struct node *attrs, int size)
{
    int i,j;
    for (i =0;i<size;i++)
    {
        for(j=i+1;j<size;j++){
            // if(attrs[i].name == NULL) attrs[i].name = (char*)malloc(1);
            if(strcmp(attrs[i].name,attrs[j].name) == 0)
            {
                printf("Same variables names not allowed %s \n",attrs[i].name);
            }
        }
    }
}

struct func_node* get_func_entry(char *name)
{
    struct func_node *act_func = func_table;
```

```
    while(act_func!=NULL)
    {
        // printf("%s is name of fn  \n",act_func->name);
        if(name == NULL) name = (char*)malloc(1);
        if(strcmp(act_func->name,name)==0)
            return act_func;
        act_func = act_func->next;
    }

    return NULL;
}

void insert_func_entry(struct node type,struct node id,struct list_attr attrs)
{
    if(get_func_entry(id.name) != NULL)
    {
        printf("Function already declared\n");
        return;
    }

    struct func_node *n = (struct func_node *)malloc(sizeof(struct func_node));
    strcpy(n->name, id.name);
    n->et = type.et;
    n->ptr_depth = type.ptr_depth;
    n->valid = 0;

    check_distinct(attrs.list_var,attrs.size);
    n->param_list = attrs.list_var;
    n->num_params = attrs.size;

    // printf("Inserting %s\n",n->name);
    n->next = func_table;
    func_table = n;
}

struct node check_func_exists(char *name)
{
    struct node ret_node;

    struct func_node *tmp = get_func_entry(name);

    strcpy(ret_node.name,name);

    if(tmp == NULL){
        ret_node.et = ERROR;
        ret_node.valid = 0;
    }
    else{
        ret_node.ptr_depth = tmp->ptr_depth;
        ret_node.size = 0;
        ret_node.et = tmp->et;
```

```
        ret_node.valid = 1;
    }

    return ret_node;
}

int match_node_list(struct node *list_var,int lsize,struct node *param_list,int psize,int isProto)
{

    if(lsize != psize)
    {
        printf("Number of var mismatch\n");
        return 0;
    }
    int i;

    for(i = 0;i<lsize;i++)
    {
        int check;
        if(isProto) check = !match_node_exact(list_var[i],param_list[i]);
        else check = !match_node(&list_var[i],&param_list[i]);
        if(check)
        {
            printf("variable mismatch : %s\n",list_var[i].name);
            return 0;
        }
    }
    printf("\n");
      return 1;
}


int check_func_proto(struct node type,struct node id,struct list_attr attrs)
{
    struct func_node *act_func = get_func_entry(id.name);

    if(act_func == NULL){
        printf("Function prototype not declared\n");
        return 0;
    }

    if(act_func->et == type.et
        && act_func->ptr_depth == type.ptr_depth
        && match_node_list(attrs.list_var,attrs.size,act_func->param_list,act_func->num_params,1))
    {
        if(act_func->valid == 1) return 0;
        act_func->valid = 1;
        return 1;
    }

    return 0;
}
```

```
int check_param_match(struct node fn,struct list_attr attrs)
{
  // printf("In check param\n");
   struct func_node *func = get_func_entry(fn.name);

   int ret_val = match_node_list(func->param_list,func->num_params,attrs.list_var,attrs.size,0);


   if(ret_val == 0)
      printf("Parameters mismatch for the Function %s\n",func->name);

   return ret_val;
}


int match_node_exact(struct node n1,struct node n2)
{
   if(n1.et == ERROR || n2.et == ERROR)
      return 0;

   if(strcmp(n1.name,n2.name)) return 0;

   if(n1.et != n2.et || n1.ptr_depth != n2.ptr_depth) return 0;
   if(n1.size != n2.size) return 0;
   int i = 0;
   for(;i<n1.size;++i ) if(n1.dim_list[i] != n2.dim_list[i]) return 0;
   return 1;
}


int match_node(struct node *n1,struct node *n2)
{
  // printf("comparing nodes %s %s %d\n",(*n1).name,(*n2).name,(*n2).et );
   struct node ret = compatible(*n1,*n2);


   if(ret.et == ERROR)
         return 0;
   struct node t1 = *n1,t2 = *n2;



   if(t1.ptr_depth || t2.ptr_depth == 0){
      int i=0;

      for(;i<t1.size;++i) {
         // printf("****%d %d *****\n", t1.dim_list[i],t2.dim_list[i]);
         if(t1.dim_list[i] != t2.dim_list[i]) return 0;
      }
```

```
    if(t2.size == 0){
        if(t1.et != CHAR_T || t1.ptr_depth > 0){
            fprintf(asmout, "addi $sp,$sp,-4\n" );
            fprintf(asmout, "sw %s,($sp)\n",t2.reg_name);
        }
        else{
            fprintf(asmout, "addi $sp,$sp,-4\n" );
            fprintf(asmout, "sw %s,($sp)\n",t2.reg_name);
        }
        freename(t2.reg_name);
    }
    else{
        int off;
        for(off = t2.st_off;off < t2.st_off + t2.prod * t2.typeSize; off += t2.typeSize){
            fprintf(asmout, "addi $a1,$a0,-%d\n",off + t2.typeSize );

            assert(t2.valid_off == 1);
            fprintf(asmout, "sub $a1,$a1,%s\n",t2.reg_off );

            if(t2.et != CHAR_T || t2.ptr_depth > 0) fprintf(asmout, "lw $a2,($a1)\n");
            else fprintf(asmout, "lw $a2,($a1)\n");

            if(t1.et != CHAR_T || t1.ptr_depth > 0){
            fprintf(asmout, "addi $sp,$sp,-4\n" );
            fprintf(asmout, "sw $a2,($sp)\n");
            }
            else{
                fprintf(asmout, "addi $sp,$sp,-4\n" );
                fprintf(asmout, "sw $a2,($sp)\n");
            }
        }
        t2.valid_off = 0;
        freename(t2.reg_off);
    }


}
else{
    int ctr=0;
    fprintf(asmout, "move $a1,%s\n",t2.reg_name );
    for(;ctr < t2.ptr_depth -1;++ctr){
        fprintf(asmout, "lw $a2,($a1)\n" );
        fprintf(asmout, "move $a1,$a2\n" );
    }

    int off = 0;
    int sz = 4;
    if(t2.et == CHAR_T) sz = 4;

    for(;off < t1.prod * sz; off += sz){

        if(sz == 4) fprintf(asmout, "lw $a2,($a1)\n");
```

```
            else fprintf(asmout, "lw $a2,($a1)\n");

            if(t1.et != CHAR_T || t1.ptr_depth > 0){
               fprintf(asmout, "addi $sp,$sp,-4\n" );
               fprintf(asmout, "sw $a2,($sp)\n");
            }

            else{
               fprintf(asmout, "addi $sp,$sp,-4\n" );
               fprintf(asmout, "sw $a2,($sp)\n");
            }

            fprintf(asmout, "addi $a1,$a1,%d\n",sz);
         }
         freename(t2.reg_name);
      }

      return 1;

}



int exists_in_scope(struct node var){
   struct node* temp = symb_table;
   while(temp){

      if(temp->scope == cur_scope && temp->valid == 1 && strcmp(temp->name,var.name) == 0)
{
         return 1;
      }
      temp = temp->next;
   }
   return 0;
}

void patch_type(struct node type,struct list_attr lis){
   int iter=0;
   for(iter = 0;iter < lis.size; ++iter){
      //printf("name of var %s %d\n",lis.list_var[iter].name,cur_scope );
      if(exists_in_scope(lis.list_var[iter])){
         printf("%s variable already defined\n",lis.list_var[iter].name);
      }
      else{
         lis.list_var[iter].ptr_depth = type.ptr_depth;
         lis.list_var[iter].et = type.et;
         lis.list_var[iter].scope = cur_scope;

         if(lis.list_var[iter].size){
            lis.list_var[iter].prod = 1;
            int i = 0;
            for(;i < lis.list_var[iter].size;++i) lis.list_var[iter].prod *= lis.list_var[iter].dim_list[i];
```

```
            }
            else lis.list_var[iter].prod = 1;

            int cprod = lis.list_var[iter].prod;
            lis.list_var[iter].st_off = cur_off;

            if(1){
               lis.list_var[iter].typeSize = 4; /// MIPS pointer size
               if(cur_scope){
                  fprintf(asmout, "addi $sp,$sp,-%d\n",4*cprod );
                  cur_off += 4*cprod;
               }
               else{
                     fprintf(asmout, "%s .space %d\n", lis.list_var[iter].name,4*cprod);
               }
            }

            insert_symb_table(lis.list_var+iter);
         }
      }
   }

struct node get_sym(char* var_name,struct list_brac* lsb){
   struct node* temp = symb_table;
   struct node* node_found = NULL;
   int cur_max_scope = -1000000;

   while(temp){
      if(temp->scope <= cur_scope && temp->valid == 1 && strcmp(temp->name,var_name) == 0)
{
         if(temp->scope > cur_max_scope){
            cur_max_scope = temp->scope;
            node_found = temp;
         }
      }
      temp = temp->next;
   }

   struct node ret;

   int size_used = 0;
   if(lsb) size_used = lsb->size;

   if(node_found == NULL || node_found -> size < size_used){
      printf("variable not found or too many boxes %s\n",var_name);
      ret.et = ERROR;

      ret.reg_name = (char*)malloc(1);
      ret.reg_off = (char*)malloc(1);
      return ret;
   }
```

```
    //ret.et = node_found->et;
    //ret.ptr_depth = node_found->ptr_depth;

    ret = *node_found;
    ret.reg_off = newname();
    fprintf(asmout, "li %s,0\n",ret.reg_off );
    ret.valid_off = 1;

    //int addoff = 0;
    if(lsb){
        int i=0;
        for(;i<size_used;++i){
            char cur_reg[4];
            strcpy(cur_reg,lsb->reg_name[i]);

            fprintf(asmout, "move $a0,%s\n",cur_reg );

            int j = 0;
            for(;j < node_found->size - size_used + i;++j) {
                fprintf(asmout, "li %s,%d\n",cur_reg,node_found->dim_list[j]);
                fprintf(asmout, "mult $a0,%s\n",cur_reg );
                fprintf(asmout, "mflo $a0\n");
            }
            freename(cur_reg);

            fprintf(asmout, "add %s,%s,$a0\n",ret.reg_off,ret.reg_off );
        }
    }
    //ret.st_off += addoff*ret.typeSize;
    fprintf(asmout, "li $a0,%d\n",ret.typeSize);
    fprintf(asmout, "mult $a0,%s\n",ret.reg_off );
    fprintf(asmout, "mflo %s\n",ret.reg_off);


    ret.size = node_found -> size - size_used;

    ret.prod = 1;

    ret.dim_list = (int*) malloc(sizeof(int)*ret.size);
    int i = 0;
    for(;i<node_found->size - size_used;++i) {
        ret.dim_list[i] = node_found->dim_list[i];
        ret.prod *= ret.dim_list[i];
    }

    return ret;
}

void delete_entries_on_ret(){
    struct node* temp = symb_table;
    while(temp){
```

```
        if(temp->scope <= cur_scope && temp -> valid == 1){
            assert(cur_scope>0);
            fprintf(asmout, "addi $sp,$sp,%d\n", temp->prod * temp->typeSize);
        }
        temp = temp->next;
    }
}
void delete_entries(){
    struct node* temp = symb_table;
    while(temp){

        if(temp->scope == cur_scope && temp -> valid == 1){
            assert(cur_scope>0);

            temp->valid = 0;
            cur_off -= temp->prod * temp->typeSize;

            fprintf(asmout, "addi $sp,$sp,%d\n", temp->prod * temp->typeSize);
        }
        temp = temp->next;
    }
}

struct node print_comp_error(struct node exp1, struct node exp2){
    struct node ret;
    ret.et = ERROR;

    ret.reg_name = (char*)malloc(1);
    ret.reg_off = (char*)malloc(1);
    printf("%d ptr: %d size: %d typecast not compatible with %d ptr: %d size:
%d\n",exp1.et,exp1.ptr_depth,exp1.size,
     exp2.et,exp2.ptr_depth,exp2.size);

    /*freename(exp1.reg_name);
    freename(exp1.reg_off);

    freename(exp2.reg_name);
    freename(exp2.reg_off);*/

    return ret;
}
struct node compatible(struct node exp1,struct node exp2){
    if(exp1.et == ERROR || exp2.et == ERROR) {
        exp1.et = ERROR;
        /*freename(exp1.reg_name);
        freename(exp1.reg_off);

        freename(exp2.reg_name);
        freename(exp2.reg_off);*/
        exp1.reg_name = (char*)malloc(1);
        exp1.reg_off = (char*)malloc(1);
```

```
        return exp1;
    }

    if(exp1.ptr_depth > 0){
        if(exp1.size != exp2.size || exp2.ptr_depth != exp1.ptr_depth) return
print_comp_error(exp1,exp2);
        exp1.expr_val = -1000000;
        return exp1;
    }
    if(exp1.size > 0){
        if(exp2.size && exp2.ptr_depth) return print_comp_error(exp1,exp2);
        if(exp2.size + exp2.ptr_depth != exp1.size) return print_comp_error(exp1,exp2);

        if(exp1.et != FLOAT_T && exp2.et == FLOAT_T) return print_comp_error(exp1,exp2);
        exp1.expr_val = -1000000;
        return exp1;
    }
    if(exp2.ptr_depth > 0 || exp2.size > 0 ) return print_comp_error(exp1,exp2);
    if(exp1.et != FLOAT_T && exp2.et == FLOAT_T) return print_comp_error(exp1,exp2);
    exp1.expr_val = -1000000;
    return exp1;
}

struct node print_opt_error(struct node exp1,enum op_type op, struct node exp2){
    struct node ret;
    ret.et = ERROR;


    ret.reg_name = (char*)malloc(1);
    ret.reg_off = (char*)malloc(1);

    printf("%d ptr: %d size: %d op: %d not compatible with %d ptr: %d size:
%d\n",exp1.et,exp1.ptr_depth,exp1.size,
    op, exp2.et,exp2.ptr_depth,exp2.size);

    /*freename(exp1.reg_name);
    freename(exp1.reg_off);

    freename(exp2.reg_name);
    freename(exp2.reg_off);*/

    return ret;
}
struct node comp_binopt_type(struct node exp1,enum op_type op, struct node exp2){
    if(exp1.et == ERROR || exp2.et == ERROR) {
        exp1.et = ERROR;

        exp1.reg_name = (char*)malloc(1);
        exp1.reg_off = (char*)malloc(1);
        return exp1;
    }
```

```
    struct node ret;
    if(op == _AND_AND || op == _OR_OR){
        if(exp1.size + exp1.ptr_depth > 0 || exp2.size + exp2.ptr_depth > 0){
            return print_opt_error(exp1,op,exp2);
        }
        ret.expr_val = -1000000;
        ret.et = BOOL;
        ret.ptr_depth = ret.size = 0;
        ret.typeSize = 4;

        if(op == _AND_AND){
            fprintf(asmout, "beq %s,$zero,NOT_LABEL%d\n",exp1.reg_name,not_label );
            fprintf(asmout, "beq %s,$zero,NOT_LABEL%d\n",exp2.reg_name,not_label );

            fprintf(asmout, "li %s,1\n",exp1.reg_name );
            fprintf(asmout, "j NOT_LABEL_END%d\n",not_label );
            fprintf(asmout, "NOT_LABEL%d:\n li %s,0\n",not_label,exp1.reg_name );
            fprintf(asmout, "NOT_LABEL_END%d:\n",not_label );
            not_label++;
            ret.reg_name = exp1.reg_name;
            freename(exp2.reg_name);
        }
        else{
            fprintf(asmout, "or %s,%s,%s\n",exp1.reg_name,exp1.reg_name,exp2.reg_name );
            freename(exp2.reg_name);

            fprintf(asmout, "bne %s,$zero,NOT_LABEL%d\n",exp1.reg_name,not_label );
            fprintf(asmout, "li %s,0\n",exp1.reg_name );
            fprintf(asmout, "j NOT_LABEL_END%d\n",not_label );
            fprintf(asmout, "NOT_LABEL%d:\n li %s,1\n",not_label,exp1.reg_name );
            fprintf(asmout, "NOT_LABEL_END%d:\n",not_label );
            not_label++;
            ret.reg_name = exp1.reg_name;
        }

        return ret;
    }
    if(op == _EEQ || op == _NEQ || op == _GR || op == _LE || op == _GRQ || op == _LEQ){
        if(exp1.size > 0 || exp2.size > 0 || exp1.ptr_depth != exp2.ptr_depth || exp1.et != exp2.et){
            return print_opt_error(exp1,op,exp2);
        }
        ret.expr_val = -1000000;
        ret.et = BOOL;
        ret.ptr_depth = ret.size = 0;
        ret.typeSize = 4;

        if(op == _EEQ) fprintf(asmout, "beq %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );
        else if(op == _NEQ) fprintf(asmout, "bne %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );
        else if(op == _GR) fprintf(asmout, "bgt %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );
```

```
        else if(op == _GRQ) fprintf(asmout, "bge %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );
        else if(op == _LE) fprintf(asmout, "blt %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );
        else if(op == _LEQ) fprintf(asmout, "ble %s,%s,NOT_LABEL
%d\n",exp1.reg_name,exp2.reg_name,not_label );

        fprintf(asmout, "li %s,0\n",exp1.reg_name );
        fprintf(asmout, "j NOT_LABEL_END%d\n",not_label );
        fprintf(asmout, "NOT_LABEL%d:\n li %s,1\n",not_label,exp1.reg_name );
        fprintf(asmout, "NOT_LABEL_END%d:\n",not_label );
        not_label++;
        ret.reg_name = exp1.reg_name;
        freename(exp2.reg_name);

        return ret;
    }




    if(op == _PLUS || op == _MINUS){
        if(exp1.ptr_depth + exp1.size > 0 && exp2.ptr_depth + exp2.size > 0) return
print_opt_error(exp1,op,exp2);
        if(exp2.size + exp2.ptr_depth > 0){
            struct node temp = exp1;
            exp1 = exp2;
            exp2 = temp;

        }



        if(exp1.ptr_depth + exp1.size > 0){
            if(exp2.et != FLOAT_T){

                if(exp1.size > 0){
                    fprintf(asmout, "addi $a0,$fp,-%d\n",exp1.st_off + exp1.typeSize );
                    fprintf(asmout, "sub $a0,$a0,%s\n",exp1.reg_off);
                    freename(exp1.reg_off);
                    exp1.valid_off = 0;

                    exp1.reg_name = newname();
                    exp1.ptr_depth += exp1.size;
                    exp1.size = 0;
                    fprintf(asmout, "move %s,$a0\n",exp1.reg_name );
                }
                if(1/*!(exp1.ptr_depth == 1 && exp1.et == CHAR_T)*/){
                    fprintf(asmout, "li $a0,4\n" );
                    fprintf(asmout, "mult $a0,%s\n",exp2.reg_name );
                    fprintf(asmout, "mflo %s\n",exp2.reg_name );
```

```
            }
            fprintf(asmout, "sub %s,%s,%s\n",exp1.reg_name,exp1.reg_name,exp2.reg_name );
            freename(exp2.reg_name);

            ret.ptr_depth = exp1.ptr_depth;
            ret.et = exp1.et;
            ret.size = 0;
            ret.expr_val = -1000000;
            ret.reg_name = exp1.reg_name;
            ret.typeSize = 4;

            return ret;
          }
          else return print_opt_error(exp1,op,exp2);
        }

        ret.size = ret.ptr_depth = 0;
        ret.typeSize = 4;
        //if(exp1.typeSize == 1 && exp2.typeSize == 1) ret.typeSize = 4;

        if(op == _PLUS){
          ret.expr_val = add(exp2.expr_val,exp1.expr_val);
          fprintf(asmout, "add %s,%s,%s\n",exp1.reg_name,exp1.reg_name,exp2.reg_name );
        }
        else {
          ret.expr_val = sub(exp1.expr_val,exp2.expr_val);
          fprintf(asmout, "sub %s,%s,%s\n",exp1.reg_name,exp1.reg_name,exp2.reg_name );
        }
        freename(exp2.reg_name);
        ret.reg_name = exp1.reg_name;

        if(exp1.et == FLOAT_T || exp2.et == FLOAT_T) ret.et = FLOAT_T;
        else {
          if((exp1.et == CHAR_T && exp2.et == INT_T) || (exp2.et == CHAR_T && exp1.et ==
INT_T)) ret.et = CHAR_T;
          else ret.et = INT_T;
        }
        return ret;
      }

      if(exp1.size + exp1.ptr_depth > 0 || exp2.size + exp2.ptr_depth > 0){
        return print_opt_error(exp1,op,exp2);
      }
      ret.size = ret.ptr_depth = 0;
      ret.typeSize = 4;
      // if(exp1.typeSize == 1 && exp2.typeSize == 1) ret.typeSize = 4;

      if(exp1.et == FLOAT_T || exp2.et == FLOAT_T) ret.et = FLOAT_T;
      else ret.et = INT_T;

      if(op == _TIMES){
        ret.expr_val = mul(exp2.expr_val,exp1.expr_val);
```

```
        fprintf(asmout, "mult %s,%s\n",exp1.reg_name,exp2.reg_name );
        fprintf(asmout, "mflo %s\n",exp1.reg_name );
    }
    else {
        ret.expr_val = _div(exp1.expr_val,exp2.expr_val);
        fprintf(asmout, "div %s,%s\n",exp1.reg_name,exp2.reg_name );
        fprintf(asmout, "mflo %s\n",exp1.reg_name );
    }
    freename(exp2.reg_name);
    ret.reg_name = exp1.reg_name;

    return ret;
}

int add(int a,int b)
{
    if(a!=-1000000 && b!=-1000000)
        return a+b;

    return -1000000;
}

int mul(int a,int b){
    if(a!=-1000000 && b!=-1000000)
        return a*b;

    return -1000000;
}

int sub(int a,int b){
    if(a!=-1000000 && b!=-1000000)
        return a-b;

    return -1000000;
}

int _div(int a,int b){
    if(a!=-1000000 && b!=-1000000){
        if(b==0){
            printf("Cannot divide by 0\n");
            return -1000000;
        }

        return a/b;
    }

    return -1000000;
}

struct node comp_opt_type(enum op_type op,struct node exp){
    if(exp.et == ERROR) {
        /*freename(exp.reg_name);
```

```
        freename(exp.reg_off);*/

        exp.reg_name = (char*)malloc(1);
        exp.reg_off = (char*)malloc(1);
        return exp;
    }

    struct node ret;
    ret.expr_val = -1000000;
    if(exp.size > 0){
        ret.et = ERROR;

        ret.reg_name = (char*)malloc(1);
        ret.reg_off = (char*)malloc(1);
        printf("%d op not compatible with %d ptr: %d size: %d
type\n",op,exp.et,exp.ptr_depth,exp.size);
        /*freename(exp.reg_name);
        freename(exp.reg_off);*/
        return ret;
    }
    if(exp.ptr_depth > 0){
        if(op == _NOT){
            ret.et = ERROR;

        ret.reg_name = (char*)malloc(1);
        ret.reg_off = (char*)malloc(1);
            /*freename(exp.reg_name);
            freename(exp.reg_off);*/
            printf("%d op not compatible with %d ptr: %d size: %d
type\n",op,exp.et,exp.ptr_depth,exp.size);
            return ret;
        }
        ret = exp;
        return ret;
    }
    if(op == _NOT){
        if(exp.et == FLOAT_T){
            ret.et = ERROR;

        ret.reg_name = (char*)malloc(1);
        ret.reg_off = (char*)malloc(1);
            /*freename(exp.reg_name);
            freename(exp.reg_off);*/
            printf("%d op not compatible with %d ptr: %d size: %d
type\n",op,exp.et,exp.ptr_depth,exp.size);
            return ret;
        }
        ret = exp;
        ret.et = BOOL;
        return ret;
    }
```

```
    if(exp.et == BOOL){
        ret.et = ERROR;

        ret.reg_name = (char*)malloc(1);
        ret.reg_off = (char*)malloc(1);
        /*freename(exp.reg_name);
        freename(exp.reg_off);*/
        printf("%d op not compatible with %d ptr: %d size: %d
type\n",op,exp.et,exp.ptr_depth,exp.size);
        return ret;
    }
    ret = exp;
    return ret;
}


char *newname()
{
    int i=0;
    for(;i<NUM_REG;++i) if(!used[i]){
        char *c = (char *)malloc(sizeof(char)*4);
        c = Names[i];
        used[i]=1;
        return c;
    }


    fprintf( stderr,": Expression too complex\n" );
    exit(1);
}

void freename(char *s)
{
    if(s == NULL) return;

    int i=0;
    for(;i<NUM_REG;++i) if(strcmp(Names[i],s) == 0){
        //assert(used[i] == 1);
        used[i]=0;
        return;
    }
}

int main (void)
{
        fout = fopen("parser.txt","w");
        asmout= fopen("asm.s","w");
        fprintf(asmout, ".data\n");

        func_table = NULL;
        symb_table = NULL;
        cur_scope=0;
```

```
        not_label = 0;

        int i=0;
        for(;i<NUM_REG;++i) {
            char temp[4];
            if(i <= 9){
                sprintf(temp,"$t%d",i);
                strcpy(Names[i],temp);
            }
            else{
                sprintf(temp,"$s%d",i-10);
                strcpy(Names[i],temp);
            }
            used[i] = 0;
        }

     //printf("dfnaksjdnkjasndkjans\n");
    // n1.reg_off = (char*)malloc(1);

            return yyparse ();
}

void yyerror (char *s) {fprintf (stderr,  "LINE:%d %s \n",mylineno,s);}
```

Flex code:

```
%{
    #include <stdio.h>
    #include "y.tab.h"
    int mylineno = 1;
%}

%%
[\n]              {mylineno++;}
[ \t]            {;}
,              {return COMMA;}
\;           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return SEMI;}
\+\+           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return PLUSPLUS      ;}
\+           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return PLUS;}
\*           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return TIMES;}
\(           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return LP      ;}
\)           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return RP                ;}
--           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return MINUSMINUS    ;}
-           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return MINUS ;}
\/           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return DIV   ;}
!=           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return NEQ   ;}
!           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return NOT   ;}
\<=           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return LEQ    ;}
\<           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return LE     ;}
\>=           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return GRQ    ;}
```

```
\>          {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return GR     ;}
==          {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return EEQ    ;}
=           {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return EQ     ;}
if          {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return IF     ;}
\{          {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return OPEN_CURL      ;}
\}          {    /*printf("Cur token %0.*s\n",yyleng,yytext);  */  return CLOSE_CURL     ;}
int                                 {return INT_TYPE;}
print                       {return PRINT;}
endl                        {return ENDL;}
float                       {return FLOAT_TYPE;}
char                        {return CHAR_TYPE;}
else          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return ELSE   ;}
main          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return MAIN   ;}
while         {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return WHILE  ;}
do          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return DO     ;}
switch          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return SWITCH ;}
case          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return CASE    ;}
default         {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return DEFAULT;}
for         {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return FOR    ;}
break         {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return BREAK  ;}
continue          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return CONTINUE       ;}
\]          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return RSQBRAC;}
\[          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return LSQBRAC;}
\.          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return DOT;}
:           {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return COLON  ;}
return          {    /*printf("Cur token %0.*s\n",yyleng,yytext);      */      return RETURN;}
[_a-zA-Z][0-9a-zA-Z_]*     {return ID;}
[0-9]+(\.[0-9]+)                  {   return FLOAT;}
[0-9]+                                    {  return INT;}
\'[^\']\'                           { return CHAR;}
\"[^\"]*\"            { return STRING; }
\&\&                                      {return AND_AND;}
\&                                        {return AMP;}
\|\|                                      {return OR_OR;}
.               {/*printf("Unexpected Cur token %0.*s ",yyleng,yytext);*/}
%%

int yywrap (void) {return 1;}
```