THE IMPLEMENTATION OF A FUNCTIONAL QUERY LANGUAGE FRONT-END TO A RELATIONAL DATABASE SYSTEM

VOL II

HANIFA UNISA SHAH

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

June 1989

LIST OF CONTENTS

VOL II

APPENDIX A: Daplex Syntax 3

APPENDIX B: Lex Specification 5

APPENDIX C: Yacc Specification 18

APPENDIX D: Programs for the FQLFE System 37

APPENDIX E: Results of Example Queries 287

APPENDIX A: Daplex Syntax

```
program = [statement]
statement = declarative | imperative
declarative = "DECLARE" funcspec ("->"|"->") expr[order]|
            "DEFINE" funcspec ("->"|"->")
                  (expr|
                  "INVERSE" "OF" funcspec)|
                  "TRANSITIVE" "OF" funcspec|
                  "COMPOUND" "OF" tuple)
                  ("INTERSECTION" | "UNION") "OF" expr {"," expr}|

                  "DIFFERENCE" "OF" expr "," expr|
                  )[order]
            "DEFINE" "CONSTAINT" funcspec "->" boolean
            "DEFINE" "TRIGGER" funcspec "->" boolean imperative
            "PERFORM" update "USING" imperative.
funcspec = funcid "("[tuple]")".
tuple = expr{"," expr}.
expr = set | singleton
set = mvfuncall|typeid|
      "{" [singleton {"," singleton}] "}".
      set "SUCH" "THAT" pred |
      set comp (singleton| quant set) |
      identifier "IN" set|
      expr "AS" typeid
      "(" set ")" | gpset.
singleton = constant|vblid|svfuncall|aggcall | pred |
      "THE" set | "A" "NEW" typeid |
      "THE" set ("PRECEEDING" | "FOLLOWING") singleton |
      "(" singleton ")" | gpsingleton
svfuncall = funcall
mvfuncall = funcall
funcall = funcid "("[tuple] ")".
aggcall = aggid "(" bag ")".
bag = expr | singleton "OVER" tuple.
pred = boolean
      "FOR" ( singleton | quant set ) pred |
      (singleton | quant set) comp (singleton | quant set)|
      quant set  "(" EXIST" | "EXISTS" ).
comp = ">" | "<" | "=" | "EQ" | "NE" | "LT" | "GT" | "LE" | "GE".
quant = "SOME" | "EVERY" | "NO" |
      ( "AT" ( "LEAST" | "MOST" )| "EXACTLY") integer.
integer = singleton
string = singleton
boolean = singleton
constant = int|str|bool
int = digit[digit]
str = """character [character] """".
bool = "TRUE" | "FALSE".
imperative = forloop | update | gpimperative.
forloop = "FOR" "EACH" set [order] imperative |
      "FOR" singleton imperative.
order = "IN" "ORDER"
      ["BY" [("ASCENDING" | "DESCENDING")]singleton).
update = "LET" svfuncall "=" singleton |
      ("LET" | "INCLUDE" | "EXCLUDE") mvfuncall "=" expr|
      "INSERT" mvfuncall "=" (singleton| set [order])
            "PRECEEDING"|"FOLLOWING") singleton.
vblid = identifier.
typeid = identifier.
funcid = identifier.
aggid = identifier.
```

APPENDIX B: Lex Specification

```
/*          FILE: Lex specification for FQLFE  */


%{

/*    includes for this file */

#include "structures.h"
#include "y.tab.h"

/*    externals for this file      */

extern int atoi();

/*    locals for this file    */

int lineno = 0;
int yycode = 0;
static int count = 0;
char ch;

char * evaluate_identifier()
{
      char *ptr;
      int *p;
      p = (int *)malloc(strlen(yytext) + 1);
      ptr = (char *) p;
      strcpy(ptr, yytext);
      return(ptr);
}

char * evaluate_string()
{
      char *ptr;
      int *p;
      p = (int *)malloc(strlen(yytext) + 1);
      ptr = (char *) p;
      strcpy(ptr, yytext);
      return(ptr);
}

evaluate_integer()
{
      int i;
      yytext[(strlen(yytext))] = '\0';
      i = atoi(yytext);
      return(i);
}


%}
digit       [0-9]
blank       [\t]

/*    lex parameters    */

%p 8000
%o 5000
%e 5000
%n 1000
%a 10000

/******
```

```
lex rules for FQLFE
******/

%%


"->>"
                    {

      yylval.VDHARR=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VDHARR->textval.s = "DHARR";
                        yylval.VDHARR->type = 1;
                        return( DHARR );
                    }
"->"
                    {

      yylval.VSHARR=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VSHARR->textval.s = "SHARR";
                        yylval.VSHARR->type = 1;
                        return( SHARR );
                    }
"INVERSE"|"inverse"
                    {

      yylval.VINVERSE=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VINVERSE->textval.s = "INVERSE";
                        yylval.VINVERSE->type = 1;
                        return( INVERSE );
                    }
"TRANSITIVE"|"transitive"
                    {

      yylval.VTRANSITIVE=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VTRANSITIVE->textval.s =
                              "TRANSITIVE";
                        yylval.VTRANSITIVE->type = 1;
                        return( TRANSITIVE );
                    }
"COMPOUND"|"compound"
                    {

      yylval.VCOMPOUND=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VCOMPOUND->textval.s = "COMPOUND";
                        yylval.VCOMPOUND->type = 1;
                        return( COMPOUND );
                    }
"INTERSECTION"|"intersection"
                    {

      yylval.VINTERSECTION=(OPR_TYPE*)malloc(sizeof(OPR_TYPE));
                        yylval.VINTERSECTION->textval.s =
                              "INTERSECTION";
                        yylval.VINTERSECTION->type = 1;
                        return( INTERSECTION );
                    }
"UNION"|"union"
                    {
                        yylval.VUNIONS =(OPR_TYPE*) malloc
                              (sizeof(OPR_TYPE));
                        yylval.VUNIONS ->textval.s = "UNIONS";
                        yylval.VUNIONS ->type = 1;
                        return( UNIONS );
```

```
                    }
"DIFFERENCE"|"difference"
                    {
                          yylval.VDIFFERENCE =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VDIFFERENCE ->textval.s =
"DIFFERENCE";
                          yylval.VDIFFERENCE ->type = 1;
                          return( DIFFERENCE );
                    }
"FOR"|"for"
                    {
                          yylval.VFOR =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VFOR ->textval.s = "FOR";
                          yylval.VFOR ->type = 1;
                          return( FOR );
                    }
"EXISTS"|"EXIST"|"exists"|"exist"
                    {
                          yylval.VEXISTS =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VEXISTS ->textval.s = "EXISTS";
                          yylval.VEXISTS ->type = 1;
                          return( EXISTS );
                    }
">"|"GT"|"gt"
                    {
                          yylval.VGTHAN =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VGTHAN ->textval.s = "GTHAN";
                          yylval.VGTHAN ->type = 1;
                          return( GTHAN );
                    }
"<"|"LT"|"lt"
                    {
                          yylval.VLTHAN =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VLTHAN ->textval.s = "LTHAN";
                          yylval.VLTHAN ->type = 1;
                          return( LTHAN );
                    }
"="|"EQ"|"eq"
                    {
                          yylval.VEQUALS =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VEQUALS ->textval.s = "EQUALS";
                          yylval.VEQUALS ->type = 1;
                          return( EQUALS );
                    }
"NE"|"ne"
                    {
                          yylval.VNOTEQUAL =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VNOTEQUAL ->textval.s = "NOTEQUAL";
                          yylval.VNOTEQUAL ->type = 1;
                          return( NOTEQUAL );
                    }
"LE"|"le"
                    {
                          yylval.VLESSEQ =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VLESSEQ ->textval.s = "LESSEQ";
```

```
                        yylval.VLESSEQ ->type = 1;
                        return( LESSEQ );
                    }
"GE"|"ge"
                    {
                        yylval.VGREATEQ =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VGREATEQ ->textval.s = "GREATEQ";
                        yylval.VGREATEQ ->type = 1;
                        return( GREATEQ );
                    }
"SOME"|"some"
                    {
                        yylval.VSOME =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VSOME ->textval.s = "SOME";
                        yylval.VSOME ->type = 1;
                        return( SOME );
                    }
"EVERY"|"every"
                    {
                        yylval.VEVERY =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VEVERY ->textval.s = "EVERY";
                        yylval.VEVERY ->type = 1;
                        return( EVERY );
                    }
"NO"|"no"
                    {
                        yylval.VNO =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VNO ->textval.s = "NO";
                        yylval.VNO ->type = 1;
                        return( NO );
                    }
"AT"|"at"
                    {
                        yylval.VAT =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VAT ->textval.s = "AT";
                        yylval.VAT ->type = 1;
                        return( AT );
                    }
"LEAST"|"least"
                    {
                        yylval.VLEAST =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VLEAST ->textval.s = "LEAST";
                        yylval.VLEAST ->type = 1;
                        return( LEAST );
                    }
"MOST"|"most"
                    {
                        yylval.VMOST =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VMOST ->textval.s = "MOST";
                        yylval.VMOST ->type = 1;
                        return( MOST );
                    }
"EXACTLY"|"exactly"
                    {
                        yylval.VEXACTLY =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
```

```
                        yylval.VEXACTLY ->textval.s = "EXACTLY";
                        yylval.VEXACTLY ->type = 1;
                        return( EXACTLY );
                    }
"EACH"|"each"
                    {
                        yylval.VEACH =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VEACH ->textval.s = "EACH";
                        yylval.VEACH ->type = 1;
                        return( EACH );
                    }
"AND"|"and"
                    {
                        yylval.VAND =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VAND ->textval.s = "AND";
                        yylval.VAND ->type = 1;
                        return( AND );
                    }
"OR"|"or"
                    {
                        yylval.VOR =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VOR ->textval.s = "OR";
                        yylval.VOR ->type = 1;
                        return( OR );
                    }
"("
                    {
                        yylval.VLPAR =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VLPAR ->textval.s = "LPAR";
                        yylval.VLPAR ->type = 1;
                        return( LPAR );
                    }
")"
                    {
                        yylval.VRPAR =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VRPAR ->textval.s = "RPAR";
                        yylval.VRPAR ->type = 1;
                        return( RPAR );
                    }
"+"
                    {
                        yylval.VPLUS =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VPLUS ->textval.s = "PLUS";
                        yylval.VPLUS ->type = 1;
                        return( PLUS );
                    }
"-"
                    {
                        yylval.VMINUS =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VMINUS ->textval.s = "MINUS";
                        yylval.VMINUS ->type = 1;
                        return( MINUS );
                    }
"*"
                    {
```

```
                        yylval.VTIMES =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VTIMES ->textval.s = "TIMES";
                        yylval.VTIMES ->type = 1;
                        return( TIMES );
                  }
"/"
                  {
                        yylval.VDIVIDE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VDIVIDE ->textval.s = "DIVIDE";
                        yylval.VDIVIDE ->type = 1;
                        return( DIVIDE );
                  }
"NOT"
                  {
                        yylval.VNOT =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VNOT ->textval.s = "NOT";
                        yylval.VNOT ->type = 1;
                        return( NOT );
                  }
"."
                  {
                              return( DOT );
                  }
"DECLARE"|"declare"
                  {
                        yylval.VDECLARE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VDECLARE ->textval.s = "DECLARE";
                        yylval.VDECLARE ->type = 1;
                        return( DECLARE );
                  }
"DEFINE"|"define"
                  {
                        yylval.VDEFINE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VDEFINE ->textval.s = "DEFINE";
                        yylval.VDEFINE ->type = 1;
                        return( DEFINE );
                  }
"OF"|"of"
                  {
                        yylval.VOF =(OPR_TYPE*)
                              malloc(sizeof(OPR_TYPE));
                        yylval.VOF ->textval.s = "OF";
                        yylval.VOF ->type = 1;
                        return( OF );
                  }
"CONSTRAINT"|"constraint"
                  {
                        yylval.VCONSTRAINT =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VCONSTRAINT ->textval.s =
                              "CONSTRAINT";
                        yylval.VCONSTRAINT ->type = 1;
                        return( CONSTRAINT );
                  }
"TRIGGER"|"trigger"
                  {
                        yylval.VTRIGGER =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
```

```
                        yylval.VTRIGGER ->textval.s = "TRIGGER";
                        yylval.VTRIGGER ->type = 1;
                        return( TRIGGER );
                    }
"USING"|"using"
                    {
                        yylval.VUSING =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VUSING ->textval.s = "USING";
                        yylval.VUSING ->type = 1;
                        return( USING );
                    }
"{"
                    {
                        yylval.VLCURLY =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VLCURLY ->textval.s = "LCURLY";
                        yylval.VLCURLY ->type = 1;
                        return( LCURLY );
                    }
"}"
                    {
                        yylval.VRCURLY =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VRCURLY ->textval.s = "RCURLY";
                        yylval.VRCURLY ->type = 1;
                        return( RCURLY );
                    }
"SUCH"|"such"
                    {
                        yylval.VSUCH =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VSUCH ->textval.s = "SUCH";
                        yylval.VSUCH ->type = 1;
                        return( SUCH );
                    }
"THAT"|"that"
                    {
                        yylval.VTHAT =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VTHAT ->textval.s = "THAT";
                        yylval.VTHAT ->type = 1;
                        return( THAT );
                    }
"IN"|"in"
                    {
                        yylval.VIN =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VIN ->textval.s = "IN";
                        yylval.VIN ->type = 1;
                        return( IN );
                    }
"AS"|"as"
                    {
                        yylval.VAS =(OPR_TYPE*)malloc
                            (sizeof(OPR_TYPE));
                        yylval.VAS ->textval.s = "AS";
                        yylval.VAS ->type = 1;
                        return( AS );
                    }
"THE"|"the"
                    {
```

```
                        yylval.VTHE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VTHE ->textval.s = "THE";
                        yylval.VTHE ->type = 1;
                        return( THE );
                  }
"NEW"|"new"
                  {
                        yylval.VNEW =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VNEW ->textval.s = "NEW";
                        yylval.VNEW ->type = 1;
                        return( NEW );
                  }
"PRECEEDING"|"preceeding"
                  {
                        yylval.VPRECEEDING =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VPRECEEDING ->textval.s =
                              "PRECEEDING";
                        yylval.VPRECEEDING ->type = 1;
                        return( PRECEEDING );
                  }
"FOLLOWING"|"following"
                  {
                        yylval.VFOLLOWING =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VFOLLOWING ->textval.s =
                              "FOLLOWING";
                        yylval.VFOLLOWING ->type = 1;
                        return( FOLLOWING );
                  }
"OVER"|"over"
                  {
                        yylval.VOVER =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VOVER ->textval.s = "OVER";
                        yylval.VOVER ->type = 1;
                        return( OVER );
                  }
"PRINT"|"print"
                  {
                        yylval.VPRINT =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VPRINT ->textval.s = "PRINT";
                        yylval.VPRINT ->type = 1;
                        return( PRINT );
                  }
","
                  {
                        yylval.VCOMMA =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VCOMMA ->textval.s = "COMMA";
                        yylval.VCOMMA ->type = 1;
                        return( COMMA );
                  }
"A"
                  {
                        yylval.VA =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VA ->textval.s = "A";
                        yylval.VA ->type = 1;
                        return( A );
```

```
                    }
"TRUE"|"true"
                    {
                          yylval.VTRUE =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VTRUE ->textval.s = "TRUE";
                          yylval.VTRUE ->type = 1;
                          return( TRUE );
                    }
"FALSE"|"false"
                    {
                          yylval.VFALSE =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VFALSE ->textval.s = "FALSE";
                          yylval.VFALSE ->type = 1;
                          return( FALSE );
                    }
"EACH"|"each"
                    {
                          yylval.VEACH =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VEACH ->textval.s = "EACH";
                          yylval.VEACH ->type = 1;
                          return( EACH );
                    }
"BY"|"by"
                    {
                          yylval.VBY =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VBY ->textval.s = "BY";
                          yylval.VBY ->type = 1;
                          return( BY );
                    }
"ASCENDING"|"ascending"
                    {
                          yylval.VASCENDING =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VASCENDING ->textval.s =
                               "ASCENDING";
                          yylval.VASCENDING ->type = 1;
                          return( ASCENDING );
                    }
"DESCENDING"|"descending"
                    {
                          yylval.VDESCENDING =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VDESCENDING ->textval.s =
                               "DESCENDING";
                          yylval.VDESCENDING ->type = 1;
                          return( DESCENDING );
                    }
"LET"|"let"
                    {
                          yylval.VLET =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
                          yylval.VLET ->textval.s = "LET";
                          yylval.VLET ->type = 1;
                          return( LET );
                    }
"INCLUDE"|"include"
                    {
                          yylval.VINCLUDE =(OPR_TYPE*)malloc
                                (sizeof(OPR_TYPE));
```

```
                        yylval.VINCLUDE ->textval.s = "INCLUDE";
                        yylval.VINCLUDE ->type = 1;
                        return( INCLUDE );
                    }
"EXCLUDE"|"exclude"
                    {
                        yylval.VEXCLUDE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VEXCLUDE ->textval.s = "EXCLUDE";
                        yylval.VEXCLUDE ->type = 1;
                        return( EXCLUDE );
                    }
"INSERT"|"insert"
                    {
                        yylval.VINSERT =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VINSERT ->textval.s = "INSERT";
                        yylval.VINSERT ->type = 1;
                        return( INSERT );
                    }
"COUNT"|"count"
                    {
                        yylval.VCOUNT=(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VCOUNT->textval.s = "COUNT";
                        yylval.VCOUNT->type = 1;
                        return( COUNT);
                    }
"MAXIMUM"|"maximum"
                    {
                        yylval.VMAXIMUM =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VMAXIMUM ->textval.s = "MAXIMUM";
                        yylval.VMAXIMUM ->type = 1;
                        return( MAXIMUM );
                    }
"MINIMUM"|"minimum"
                    {
                        yylval.VMINIMUM =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VMINIMUM ->textval.s = "MINIMUM";
                        yylval.VMINIMUM ->type = 1;
                        return( MINIMUM );
                    }
"TOTAL"|"total"
                    {
                        yylval.VTOTAL =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VTOTAL ->textval.s = "TOTAL";
                        yylval.VTOTAL ->type = 1;
                        return( TOTAL );
                    }
"AVERAGE"|"AVERAGE"
                    {
                        yylval.VAVERAGE =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VAVERAGE ->textval.s = "AVERAGE";
                        yylval.VAVERAGE ->type = 1;
                        return( AVERAGE );
                    }
"ENTITY"|"entity"
                    {
```

```
                        yylval.VENTITY =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VENTITY ->textval.s = "ENTITY";
                        yylval.VENTITY ->type = 1;
                        return( ENTITY );
                  }
"INTEGER"|"integer"|"INT"|"int"
                  {
                        yylval.VINT =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VINT ->textval.s = "INT";
                        yylval.VINT ->type = 1;
                        return( INT );
                  }
"STRING"|"string"|"STR"|"str"
                  {
                        yylval.VSTR =(OPR_TYPE*)malloc
                              (sizeof(OPR_TYPE));
                        yylval.VSTR ->textval.s = "STR";
                        yylval.VSTR ->type = 1;
                        return( STR );
                  }
"-"?[0-9]+
                  {
                        yylval.VINTEGER=(INTEGER_TYPE*)malloc
                              (sizeof(INTEGER_TYPE));
                        yylval.VINTEGER->textval.d =
                              evaluate_integer();
                        yylval.VINTEGER->type = 3;
                        return( INTEGER );
                  }
"'"[^\'\n]+"'"|\"[^\"\n]+\"
                  {
                        yylval.VSTRING =(STRING_TYPE*)malloc
                              (sizeof(STRING_TYPE));
                        yylval.VSTRING ->textval.s =
                              evaluate_string ();
                        yylval.VSTRING ->type = 4;
                        return( STRING );
                  }
[A-Za-z][_0-9A-Za-z]*"("
                  {
                        yylval.VIDENTIFIERF=(IDENTIFIERF_TYPE*)
                              malloc(sizeof(IDENTIFIERF_TYPE));
                        unput('(');
                        yytext[(strlen(yytext)) - 1] = '\0';
                        yylval.VIDENTIFIERF ->textval.s =
                              evaluate_identifier();
                        yylval.VIDENTIFIERF ->type = 6;
                        return( IDENTIFIERF );
                  }
[A-Za-z][_0-9A-Za-z]*")"|[A-Za-z][_0-9A-Za-z]*","|
      [A-Za-z][_0-9A-Za-z]*"="
                  {
                        yylval.VIDENTIFIERV =(IDENTIFIERV_TYPE*)
                              malloc(sizeof(IDENTIFIERV_TYPE));
                        unput(yytext[((strlen(yytext)) - 1)]);
                        yytext[(strlen(yytext)) - 1] = '\0';
                        yylval.VIDENTIFIERV ->textval.s =
                              evaluate_identifier();
                        yylval.VIDENTIFIERV ->type = 5;
                        return( IDENTIFIERV );
                  }
```

```
[A-Za-z][_0-9A-Za-z]*
                    {
                          yylval.VIDENTIFIER =(IDENTIFIER_TYPE*)
                               malloc(sizeof(IDENTIFIER_TYPE));
                          yylval.VIDENTIFIER ->textval.s =
                               evaluate_identifier ();
                          yylval.VIDENTIFIER ->type = 2;
                          return( IDENTIFIER );
                    }
"\n"                      { lineno ++; }
[ \t]                     { ; }
[^\t\n]                   { printf("TOKEN RUBBISH is %s\n", yytext);
return( RUBBISH ); }

%%

/*    main lex routine  */
int yywrap()
    { return(1); }
```

APPENDIX C: Yacc Specification

```
/*          FILE: Yacc specification for FQLFE */

%{

/*    includes for this file */

#include          "structures.h"

/*    locals for this file    */

PROGRAM_TYPE      tree;

/*    externals for this file      */

extern int        phase;

%}

/*    Types associated with grammar symbols    */


%union
{
      OPR_TYPE          *VDHARR;
      OPR_TYPE          *VSHARR;
      OPR_TYPE          *VINVERSE;
      OPR_TYPE          *VTRANSITIVE;
      OPR_TYPE          *VCOMPOUND;
      OPR_TYPE          *VINTERSECTION;
      OPR_TYPE          *VUNIONS;
      OPR_TYPE          *VDIFFERENCE;
      OPR_TYPE          *VFOR;
      OPR_TYPE          *VEXISTS;
      OPR_TYPE          *VGTHAN;
      OPR_TYPE          *VLTHAN;
      OPR_TYPE          *VEQUALS;
      OPR_TYPE          *VNOTEQUAL;
      OPR_TYPE          *VLESSEQ;
      OPR_TYPE          *VGREATEQ;
      OPR_TYPE          *VSOME;
      OPR_TYPE          *VEVERY;
      OPR_TYPE          *VNO;
      OPR_TYPE          *VAT;
      OPR_TYPE          *VLEAST;
      OPR_TYPE          *VMOST;
      OPR_TYPE          *VEXACTLY;
      OPR_TYPE          *VEACH;
      OPR_TYPE          *VAND;
      OPR_TYPE          *VOR;
      OPR_TYPE          *VLPAR;
      OPR_TYPE          *VRPAR;
      OPR_TYPE          *VDOT;
      OPR_TYPE          *VDECLARE;
      OPR_TYPE          *VDEFINE;
      OPR_TYPE          *VOF;
      OPR_TYPE          *VCONSTRAINT;
      OPR_TYPE          *VTRIGGER;
      OPR_TYPE          *VPERFORM;
      OPR_TYPE          *VUSING;
      OPR_TYPE          *VLCURLY;
      OPR_TYPE          *VRCURLY;
      OPR_TYPE          *VSUCH;
      OPR_TYPE          *VTHAT;
```

```
OPR_TYPE           *VIN;
OPR_TYPE           *VAS;
OPR_TYPE           *VTHE;
OPR_TYPE           *VNEW;
OPR_TYPE           *VPRECEEDING;
OPR_TYPE           *VFOLLOWING;
OPR_TYPE           *VOVER;
OPR_TYPE           *VTRUE;
OPR_TYPE           *VFALSE;
OPR_TYPE           *VORDER;
OPR_TYPE           *VBY;
OPR_TYPE           *VASCENDING;
OPR_TYPE           *VDESCENDING;
OPR_TYPE           *VLET;
OPR_TYPE           *VINCLUDE;
OPR_TYPE           *VEXCLUDE;
OPR_TYPE           *VINSERT;
OPR_TYPE           *VCOMMA;
OPR_TYPE           *VA;
OPR_TYPE           *VPRINT;
OPR_TYPE           *VCOUNT;
OPR_TYPE           *VMAXIMUM;
OPR_TYPE           *VMINIMUM;
OPR_TYPE           *VTOTAL;
OPR_TYPE           *VAVERAGE;
OPR_TYPE           *VTRUE;
OPR_TYPE           *VFALSE;
OPR_TYPE           *VHAS;
OPR_TYPE           *VENTITY;
OPR_TYPE           *VINT;
OPR_TYPE           *VSTR;
OPR_TYPE           *VPLUS;
OPR_TYPE           *VMINUS;
OPR_TYPE           *VTIMES;
OPR_TYPE           *VDIVIDE;
OPR_TYPE           *VNOT;
IDENTIFIER_TYPE    *VIDENTIFIER;
IDENTIFIERV_TYPE   *VIDENTIFIERV;
IDENTIFIERF_TYPE   *VIDENTIFIERF;
INTEGER_TYPE       *VINTEGER;
STRING_TYPE        *VSTRING;
PROGRAM_TYPE       Vprogram;
STATEMENTS_TYPE    Vstatements;
STATEMENT_TYPE     Vstatement;
DECLARATIVE_TYPE   Vdeclarative;
SIMPLE_DECL_TYPE   Vsimple_decl;
COMPLEX_DECL_TYPE  Vcomplex_decl;
DEFINETYPES_TYPE   Vdefinetypes;
ARROW_TYPE         Varrow;
FUNCSPEC_TYPE      Vfuncspec;
ARGLIST_TYPE       Varglist;
MTUPLE_TYPE        Vmtuple;
STUPLE_TYPE        Vstuple;
EXPR_TYPE          Vexpr;
IMPERATIVE_TYPE    Vimperative;
FORLOOP_TYPE       Vforloop;
GPIMPERATIVE_TYP   Vgpimperative;
SET_TYPE           Vset;
SINGLETON_TYPE     Vsingleton;
OREXP1_TYPE        Vorexp1;
EXP1_TYPE          Vexp1;
ANDEXP2_TYPE       Vandexp2;
EXP2_TYPE          Vexp2;
```

```
        EXP3_TYPE           Vexp3;
        EXP4_TYPE           Vexp4;
        EXP5_TYPE           Vexp5;
        EXP6_TYPE           Vexp6;
        EXP7_TYPE           Vexp7;
        AGGCALL_TYPE        Vaggcall;
        COMP_TYPE           Vcomp;
        QUANT_TYPE          Vquant;
        LEASTMOST_TYPE      Vleastmost;
        ADDOP_TYPE          Vaddop;
        MULOP_TYPE          Vmulop;
        CONSTANT_TYPE       Vconstant;
        BOOL_TYPE           Vbool;
        FUNCID_TYPE         Vfuncid;
        TYPEID_TYPE         Vtypeid;
        VBLID_TYPE          Vvblid;
        SINGLIST_TYPE       Vsinglist;
        SVFUNCALL_TYPE      Vsvfuncall;
        MVFUNCALL_TYPE      Vmvfuncall;
        PRED_TYPE           Vpred;
}

/*    operator  tokens  */

%token DHARR
%token SHARR
%token INVERSE
%token TRANSITIVE
%token COMPOUND
%token INTERSECTION
%token UNIONS
%token DIFFERENCE
%token FOR
%token EXISTS
%token GTHAN
%token LTHAN
%token EQUALS
%token NOTEQUAL
%token LESSEQ
%token GREATEQ
%token SOME
%token EVERY
%token NO
%token AT
%token LEAST
%token MOST
%token EXACTLY
%token EACH
%token AND
%token OR
%token PLUS
%token MINUS
%token TIMES
%token DIVIDE
%token NOT

/*    lexeme   tokens        */

%token INTEGER
%token IDENTIFIER
%token IDENTIFIERV
%token IDENTIFIERF
%token STRING
```

```
/*    keyword tokens          */

%token LPAR
%token RPAR
%token DOT
%token DECLARE
%token DEFINE
%token OF
%token CONSTRAINT
%token TRIGGER
%token PERFORM
%token USING
%token LCURLY
%token RCURLY
%token SUCH
%token THAT
%token IN
%token AS
%token THE
%token NEW
%token PRECEEDING
%token FOLLOWING
%token OVER
%token TRUE
%token FALSE
%token ORDER
%token BY
%token ASCENDING
%token DESCENDING
%token LET
%token INCLUDE
%token EXCLUDE
%token INSERT
%token COMMA
%token A
%token PRINT
%token RUBBISH
%token COUNT
%token MAXIMUM
%token MINIMUM
%token TOTAL
%token AVERAGE
%token TRUE
%token FALSE
%token HAS
%token ENTITY
%token INT
%token STR

/*    Type declaration        */

%type <VTOTAL>          TOTAL
%type <VDHARR>          DHARR
%type <VSHARR>          SHARR
%type <VINVERSE>        INVERSE
%type <VTRANSITIVE>     TRANSITIVE
%type <VCOMPOUND>       COMPOUND
%type <VINTERSECTION>   INTERSECTION
%type <VUNIONS>         UNIONS
%type <VDIFFERENCE>     DIFFERENCE
%type <VFOR>            FOR
%type <VEXISTS>         EXISTS
```

```
%type <VGTHAN>          GTHAN
%type <VLTHAN>          LTHAN
%type <VEQUALS>         EQUALS
%type <VNOTEQUAL>       NOTEQUAL
%type <VLESSEQ>         LESSEQ
%type <VGREATEQ>        GREATEQ
%type <VSOME>           SOME
%type <VEVERY>          EVERY
%type <VNO>             NO
%type <VAT>             AT
%type <VLEAST>          LEAST
%type <VMOST>           MOST
%type <VEXACTLY>        EXACTLY
%type <VEACH>           EACH
%type <VAND>            AND
%type <VOR>             OR
%type <VLPAR>           LPAR
%type <VRPAR>           RPAR
%type <VDOT>            DOT
%type <VDECLARE>        DECLARE
%type <VDEFINE>         DEFINE
%type <VOF>             OF
%type <VCONSTRAINT>     CONSTRAINT
%type <VTRIGGER>        TRIGGER
%type <VPERFORM>        PERFORM
%type <VUSING>          USING
%type <VLCURLY>         LCURLY
%type <VRCURLY>         RCURLY
%type <VSUCH>           SUCH
%type <VTHAT>           THAT
%type <VIN>             IN
%type <VAS>             AS
%type <VTHE>            THE
%type <VNEW>            NEW
%type <VPRECEEDING>     PRECEEDING
%type <VFOLLOWING>      FOLLOWING
%type <VOVER>           OVER
%type <VTRUE>           TRUE
%type <VFALSE>          FALSE
%type <VORDER>          ORDER
%type <VBY>             BY
%type <VASCENDING>      ASCENDING
%type <VDESCENDING>     DESCENDING
%type <VLET>            LET
%type <VINCLUDE>        INCLUDE
%type <VEXCLUDE>        EXCLUDE
%type <VINSERT>         INSERT
%type <VCOMMA>          COMMA
%type <VA>              A
%type <VPRINT>          PRINT
%type <VTRUE>           TRUE
%type <VFALSE>          FALSE
%type <VHAS>            HAS
%type <VIDENTIFIER>     IDENTIFIER
%type <VIDENTIFIERV>    IDENTIFIERV
%type <VIDENTIFIERF>    IDENTIFIERF
%type <VINTEGER>        INTEGER
%type <VMINIMUM>        MINIMUM
%type <VSTRING>         STRING
%type <VCOUNT>          COUNT
%type <VMAXIMUM>        MAXIMUM
%type <VAVERAGE>        AVERAGE
%type <VENTITY>         ENTITY
```

```
%type <VINT>            INT
%type <VSTR>            STR
%type <VPLUS>           PLUS
%type <VMINUS>          MINUS
%type <VTIMES>          TIMES
%type <VDIVIDE>         DIVIDE
%type <VNOT>            NOT
%type <Vprogram>        program
%type <Vstatements>     statements
%type <Vstatement>      statement
%type <Vdeclarative>    declarative
%type <Vsimple_decl>    simple_decl
%type <Vcomplex_decl>   complex_decl
%type <Vdefinetypes>    definetypes
%type <Varrow>          arrow
%type <Vfuncspec>       funcspec
%type <Varglist>        arglist
%type <Vmtuple>         mtuple
%type <Vstuple>         stuple
%type <Vexpr>           expr
%type <Vimperative>     imperative
%type <Vforloop>        forloop
%type <Vgpimperative>   gpimperative
%type <Vset>            set
%type <Vsingleton>      singleton
%type <Vorexp1>         orexp1
%type <Vexp1>           exp1
%type <Vandexp2>        andexp2
%type <Vexp2>           exp2
%type <Vexp3>           exp3
%type <Vexp4>           exp4
%type <Vexp5>           exp5
%type <Vaggcall>        aggcall
%type <Vcomp>           comp
%type <Vquant>          quant
%type <Vleastmost>      leastmost
%type <Vconstant>       constant
%type <Vbool>           bool
%type <Vfuncid>         funcid
%type <Vtypeid>         typeid
%type <Vvblid>          vblid
%type <Vsinglist>       singlist
%type <Vmvfuncall>      mvfuncall
%type <Vsvfuncall>      svfuncall
%type <Vpred>           pred
%type <Vaddop>          addop
%type <Vmulop>          mulop
%type <Vexp6>           exp6
%type <Vexp7>           exp7

%start program
%%

/******
Yacc rules for FQLFE system
******/

program             :statements
                    {
                          $$ = (PROGRAM_TYPE)node2(1,$1);
                          tree = $$;
                          writeout_statements();
                    }
```

```
                ;

statements      :statement
                {
                        $$ = (STATEMENTS_TYPE)node2(1,$1);
                }
                |statements statement
                {
                        $$ = (STATEMENTS_TYPE)node3(2,$1,$2);
                }
                ;

statement       :declarative  DOT
                {
                        $$ = (STATEMENT_TYPE)node2(1,$1);
                }
                |imperative DOT
                {
                        $$ = (STATEMENT_TYPE)node2(2,$1);
                }
                |error DOT
                        {yyerrok;}
                ;

declarative     :simple_decl
                {
                        $$ = (DECLARATIVE_TYPE)node2(1,$1);
                }
                |complex_decl
                {
                        $$ = (DECLARATIVE_TYPE)node2(2,$1);
                }
                ;

simple_decl     :DECLARE funcid LPAR RPAR arrow ENTITY
                {
                        $$ = (SIMPLE_DECL_TYPE)
                              node7(1,$1,$2,$3,$4,$5,$6);
                }
                |DECLARE funcid LPAR vblid RPAR arrow INT
                {
                        $$ = (SIMPLE_DECL_TYPE)
                              node8(2,$1,$2,$3,$4,$5,$6,$7);
                }
                |DECLARE funcid LPAR vblid RPAR arrow STR
                {
                        $$ = (SIMPLE_DECL_TYPE)
                              node8(3,$1,$2,$3,$4,$5,$6,$7);
                }
                ;

complex_decl    :DEFINE funcspec arrow definetypes
                {
                        $$ = (COMPLEX_DECL_TYPE)
                              node5(1,$1,$2,$3,$4);
                }
                ;

definetypes     :INVERSE OF funcspec
                {
                        $$ = (DEFINETYPES_TYPE)node4(1,$1,$2,$3);
                }
                |TRANSITIVE OF expr
```

```
                {
                        $$ = (DEFINETYPES_TYPE)node4(2,$1,$2,$3);
                }
                |COMPOUND OF mtuple
                {
                        $$ = (DEFINETYPES_TYPE)node4(3,$1,$2,$3);
                }
                |INTERSECTION OF mtuple
                {
                        $$ = (DEFINETYPES_TYPE)node4(4,$1,$2,$3);
                }
                |UNIONS OF mtuple
                {
                        $$ = (DEFINETYPES_TYPE)node4(5,$1,$2,$3);
                }
                |DIFFERENCE OF mtuple
                {
                        $$ = (DEFINETYPES_TYPE)node4(6,$1,$2,$3);
                }
                |expr
                {
                        $$ = (DEFINETYPES_TYPE)node2(7,$1);
                }
                ;

arrow           :SHARR
                {
                        $$ = (ARROW_TYPE)node2(1,$1);
                }
                |DHARR
                {
                        $$ = (ARROW_TYPE)node2(2,$1);
                }
                ;

funcspec        :funcid LPAR mtuple RPAR
                {
                        $$ = (FUNCSPEC_TYPE)node5(1,$1,$2,$3,$4);
                }
                ;

arglist         :vblid
                {
                        $$ = (ARGLIST_TYPE)node2(1,$1);
                }
                |arglist COMMA vblid
                {
                        $$ = (ARGLIST_TYPE)node4(2,$1,$2,$3);
                }
                ;

mtuple          :expr
                {
                        $$ = (MTUPLE_TYPE)node2(1,$1);
                }
                |mtuple COMMA expr
                {
                        $$ = (MTUPLE_TYPE)node4(2,$1,$2,$3);
                }
                ;

singlist        :COMMA singleton
```

```
                    {
                          $$ = (SINGLIST_TYPE)node3(1,$1,$2);
                    }
                    |singlist COMMA singleton
                    {
                          $$ = (SINGLIST_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

stuple              :singleton
                    {
                          $$ = (STUPLE_TYPE)node2(1,$1);
                    }
                    |stuple COMMA singleton
                    {
                          $$ = (STUPLE_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

expr                :set
                    {
                          $$ = (EXPR_TYPE)node2(1,$1);
                    }
                    |singleton
                    {
                          $$ = (EXPR_TYPE)node2(2,$1);
                    }
                    ;

imperative          :forloop
                    {
                          $$ = (IMPERATIVE_TYPE)node2(1,$1);
                    }
                    |gpimperative
                    {
                          $$ = (IMPERATIVE_TYPE)node2(2,$1);
                    }
                    ;

forloop             :FOR EACH set imperative
                    {
                          $$ = (FORLOOP_TYPE)node5(1,$1,$2,$3,$4);
                    }
                    |FOR singleton imperative
                    {
                          $$ = (FORLOOP_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

gpimperative        :PRINT stuple
                    {
                          $$ = (GPIMPERATIVE_TYPE)node3(1,$1,$2);
                    }
                    ;

set                 :typeid
                    {
                          $$ = (SET_TYPE)node2(1,$1);
                    }
                    |mvfuncall
                    {
                          $$ = (SET_TYPE)node2(2,$1);
                    }
```

```
                    |LCURLY stuple RCURLY
                    {
                          $$ = (SET_TYPE)node4(3,$1,$2,$3);
                    }
                    |LPAR set RPAR
                    {
                          $$ = (SET_TYPE)node4(4,$1,$2,$3);
                    }
                    |IDENTIFIER IN set
                    {
                          $$ = (SET_TYPE)node4(5,$1,$2,$3);
                    }
                    |set AS typeid
                    {
                          $$ = (SET_TYPE)node4(6,$1,$2,$3);
                    }
                    |set comp singleton
                    {
                          $$ = (SET_TYPE)node4(7,$1,$2,$3);
                    }
                    |set comp quant set
                    {
                          $$ = (SET_TYPE)node5(8,$1,$2,$3,$4);
                    }
                    |set SUCH THAT pred
                    {
                          $$ = (SET_TYPE)node5(9,$1,$2,$3,$4);
                    }
                    ;

pred                :FOR singleton pred
                    {
                          $$ = (PRED_TYPE)node4(1,$1,$2,$3);
                    }
                    |singleton
                    {
                          $$ = (PRED_TYPE)node2(2,$1);
                    }
                    |FOR quant set pred
                    {
                          $$ = (PRED_TYPE)node5(3,$1,$2,$3,$4);
                    }
                    |singleton comp singleton
                    {
                          $$ = (PRED_TYPE)node4(4,$1,$2,$3);
                    }
                    |singleton comp quant set
                    {
                          $$ = (PRED_TYPE)node5(5,$1,$2,$3,$4);
                    }
                    |quant set comp quant set
                    {
                          $$ = (PRED_TYPE)node6(6,$1,$2,$3,$4,$5);
                    }
                    ;

singleton           :expl
                    {
                          $$ = (SINGLETON_TYPE)node2(1,$1);
                    }
                    |expl orexpl
                    {
                          $$ = (SINGLETON_TYPE)node3(2,$1,$2);
```

```
                    }
                    ;

orexp1              :OR exp1
                    {
                          $$ = (OREXP1_TYPE)node3(1,$1,$2);
                    }
                    |orexp1 OR exp1
                    {
                          $$ = (OREXP1_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

exp1                :exp2
                    {
                          $$ = (EXP1_TYPE)node2(1,$1);
                    }
                    |exp2 andexp2
                    {
                          $$ = (EXP1_TYPE)node3(2,$1,$2);
                    }
                    ;

andexp2             :AND exp2
                    {
                          $$ = (ANDEXP2_TYPE)node3(1,$1,$2);
                    }
                    |andexp2  AND exp2
                    {
                          $$ = (ANDEXP2_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

exp2                :exp3
                    {
                          $$ = (EXP2_TYPE)node2(1,$1);
                    }
                    |NOT exp3
                    {
                          $$ = (EXP2_TYPE)node3(2,$1,$2);
                    }
                    ;

exp3                :exp4
                    {
                          $$ = (EXP3_TYPE)node2(1,$1);
                    }
                    |exp4 comp exp4
                    {
                          $$ = (EXP3_TYPE)node4(2,$1,$2,$3);
                    }
                    ;

exp4                :exp5
                    {
                          $$ = (EXP4_TYPE)node2(1,$1);
                    }
                    |exp5 addop exp5
                    {
                          $$ = (EXP4_TYPE)node4(2,$1,$2,$3);
                    }
                    ;
```

```
exp5            :exp6
                {
                      $$ = (EXP5_TYPE)node2(1,$1);
                }
                |exp6 mulop exp6
                {
                      $$ = (EXP5_TYPE)node4(2,$1,$2,$3);
                }
                ;

exp6            :exp7
                {
                      $$ = (EXP6_TYPE)node2(1,$1);
                }
                |exp7 AS typeid
                {
                      $$ = (EXP6_TYPE)node4(2,$1,$2,$3);
                }
                ;

exp7            :constant
                {
                      $$ = (EXP7_TYPE)node2(1,$1);
                }
                |vblid
                {
                      $$ = (EXP7_TYPE)node2(2,$1);
                }
                |mvfuncall
                {
                      $$ = (EXP7_TYPE)node2(3,$1);
                }
                |aggcall
                {
                      $$ = (EXP7_TYPE)node2(4,$1);
                }
                |quant set HAS singleton
                {
                      $$ = (EXP7_TYPE)node5(5,$1,$2,$3,$4);
                }
                |THE set
                {
                      $$ = (EXP7_TYPE)node3(6,$1,$2);
                }
                |LPAR singleton RPAR
                {
                      $$ = (EXP7_TYPE)node4(7,$1,$2,$3);
                }
                ;

aggcall         :COUNT LPAR set RPAR
                {
                      $$ = (AGGCALL_TYPE)node5(1,$1,$2,$3,$4);
                }
                 |MAXIMUM LPAR set RPAR
                {
                      $$ = (AGGCALL_TYPE)node5(2,$1,$2,$3,$4);
                }
                |MINIMUM LPAR set RPAR
                {
                      $$ = (AGGCALL_TYPE)node5(3,$1,$2,$3,$4);
                }
                |TOTAL LPAR singleton OVER mtuple RPAR
```

```
                {
                        $$ = (AGGCALL_TYPE)
                              node7(4,$1,$2,$3,$4,$5,$6);
                }
                |AVERAGE LPAR singleton OVER mtuple RPAR
                {
                        $$ = (AGGCALL_TYPE)
                              node7(5,$1,$2,$3,$4,$5,$6);
                }
                |AVERAGE LPAR singleton RPAR
                {
                        $$ = (AGGCALL_TYPE)node5(6,$1,$2,$3,$4);
                }
                |COUNT LPAR singleton OVER mtuple RPAR
                {
                        $$ = (AGGCALL_TYPE)
                              node7(7,$1,$2,$3,$4,$5,$6);
                }
                ;

comp            :GTHAN
                {
                        $$ = (COMP_TYPE)node2(1,$1);
                }
                |LTHAN
                {
                        $$ = (COMP_TYPE)node2(2,$1);
                }
                |EQUALS
                {
                        $$ = (COMP_TYPE)node2(3,$1);
                }
                |NOTEQUAL
                {
                        $$ = (COMP_TYPE)node2(4,$1);
                }
                |LESSEQ
                {
                        $$ = (COMP_TYPE)node2(5,$1);
                }
                |GREATEQ
                {
                        $$ = (COMP_TYPE)node2(6,$1);
                }
                ;

quant           :SOME
                {
                        $$ = (QUANT_TYPE)node2(1,$1);
                }
                |EVERY
                {
                        $$ = (QUANT_TYPE)node2(2,$1);
                }
                |NO
                {
                        $$ = (QUANT_TYPE)node2(3,$1);
                }
                |AT leastmost singleton
                {
                        $$ = (QUANT_TYPE)node4(4,$1,$2,$3);
                }
                |EXACTLY singleton
```

```
                {
                        $$ = (QUANT_TYPE)node3(5,$1,$2);
                }
                ;

leastmost       :LEAST
                {
                        $$ = (LEASTMOST_TYPE)node2(1,$1);
                }
                |MOST
                {
                        $$ = (LEASTMOST_TYPE)node2(2,$1);
                }
                ;

addop           :PLUS
                {
                        $$ = (ADDOP_TYPE)node2(1,$1);
                }
                |MINUS
                {
                        $$ = (ADDOP_TYPE)node2(2,$1);
                }
                ;

mulop           :TIMES
                {
                        $$ = (MULOP_TYPE)node2(1,$1);
                }
                |DIVIDE
                {
                        $$ = (MULOP_TYPE)node2(2,$1);
                }
                ;

constant        :INTEGER
                {
                        $$ = (CONSTANT_TYPE)node2(1,$1);
                }
                |STRING
                {
                        $$ = (CONSTANT_TYPE)node2(2,$1);
                }
                |bool
                {
                        $$ = (CONSTANT_TYPE)node2(3,$1);
                }
                ;

bool            :TRUE
                {
                        $$ = (BOOL_TYPE)node2(1,$1);
                }
                |FALSE
                {
                        $$ = (BOOL_TYPE)node2(2,$1);
                }
                ;


funcid          :IDENTIFIERF
                {
                        $$ = (FUNCID_TYPE)node2(1,$1);
```

```
                    }
                    ;

typeid              :IDENTIFIER
                    {
                          $$ = (TYPEID_TYPE)node2(1,$1);
                    }
                    ;

vblid               :IDENTIFIERV
                    {
                          $$ = (VBLID_TYPE)node2(1,$1);
                    }
                    ;

svfuncall           :IDENTIFIERF LPAR singleton RPAR
                    {
                          $$ = (SVFUNCALL_TYPE)node5(1,$1,$2,$3,$4);
                    }
                    |IDENTIFIERF LPAR singleton singlist RPAR
                    {
                          $$ = (SVFUNCALL_TYPE)
                                node6(2,$1,$2,$3,$4, $5);
                    }
                    ;

mvfuncall           :IDENTIFIERF LPAR mtuple RPAR
                    {
                          $$ = (MVFUNCALL_TYPE)node5(1,$1,$2,$3,$4);
                    }
                    ;

%%
extern char *malloc();

char *node1(a)
int a;
{
      int *p;
      char *Node_Pt;

      Node_Pt = malloc(sizeof(int));
      p = (int *)Node_Pt;
      *p = a;
      return(Node_Pt);
}

char *node2(a,b)
int a,b;
{
      int *p;
      char *Node_Pt;

      Node_Pt = malloc( 2 * sizeof(int));
      p = (int *)Node_Pt;
      *p = a;
      p++;
      *p = b;

      return(Node_Pt);
}

char *node3(a,b,c)
```

```
int a,b,c;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 3 * sizeof(int));
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;

        return(Node_Pt);
}

char *node4(a,b,c,d)
int a,b,c,d;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 4 * sizeof(int));
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;
        p++;
        *p = d;

        return(Node_Pt);
}

char *node5(a,b,c,d,e)
int a,b,c,d,e;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 5 * sizeof(int));
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;
        p++;
        *p = d;
        p++;
        *p = e;

        return(Node_Pt);
}

char *node6(a,b,c,d,e,f)
int a,b,c,d,e,f;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 6 * sizeof(int));
```

```
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;
        p++;
        *p = d;
        p++;
        *p = e;
        p++;
        *p = f;

        return(Node_Pt);
}

char *node7(a,b,c,d,e,f,g)
int a,b,c,d,e,f,g;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 7 * sizeof(int));
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;
        p++;
        *p = d;
        p++;
        *p = e;
        p++;
        *p = f;
        p++;
        *p = g;

        return(Node_Pt);
}

char *node8(a,b,c,d,e,f,g,h)
int a,b,c,d,e,f,g,h;
{
        int *p;
        char *Node_Pt;

        Node_Pt = malloc( 8 * sizeof(int));
        p = (int *)Node_Pt;
        *p = a;
        p++;
        *p = b;
        p++;
        *p = c;
        p++;
        *p = d;
        p++;
        *p = e;
        p++;
        *p = f;
        p++;
        *p = g;
        p++;
```

```
        *p = h;

        return(Node_Pt);
}

/******
This routine calls the traversal routines
******/

writeout_statements()
{

        phase = 0;
        traverse_program(tree);
}
```

APPENDIX D: Programs for the FQLFE System

```
/*          FILE: structures.h          */

/* This is the definition of the major data structures used in
FQLFE */

#define NIL       0
#define STRUCT    struct
#define UNION     union

typedef int*      PTR;

typedef struct program_type         * PROGRAM_TYPE;
typedef struct statements_type      * STATEMENTS_TYPE;
typedef struct statement_type       * STATEMENT_TYPE;
typedef struct declarative_type     * DECLARATIVE_TYPE;
typedef struct simple_decl_type     * SIMPLE_DECL_TYPE;
typedef struct complex_decl_type    * COMPLEX_DECL_TYPE;
typedef struct definetypes_type     * DEFINETYPES_TYPE;
typedef struct arrow_type           * ARROW_TYPE;
typedef struct funcspec_type        * FUNCSPEC_TYPE;
typedef struct arglist_type         * ARGLIST_TYPE;
typedef struct mtuple_type          * MTUPLE_TYPE;
typedef struct stuple_type          * STUPLE_TYPE;
typedef struct expr_type            * EXPR_TYPE;
typedef struct imperative_type      * IMPERATIVE_TYPE;
typedef struct forloop_type         * FORLOOP_TYPE;
typedef struct gpimperative_type    * GPIMPERATIVE_TYPE;
typedef struct set_type             * SET_TYPE;
typedef struct setop_type           * SETOP_TYPE;
typedef struct singleton_type       * SINGLETON_TYPE;
typedef struct orexp1_type          * OREXP1_TYPE;
typedef struct exp1_type            * EXP1_TYPE;
typedef struct andexp2_type         * ANDEXP2_TYPE;
typedef struct exp2_type            * EXP2_TYPE;
typedef struct exp3_type            * EXP3_TYPE;
typedef struct exp4_type            * EXP4_TYPE;
typedef struct exp5_type            * EXP5_TYPE;
typedef struct exp6_type            * EXP6_TYPE;
typedef struct exp7_type            * EXP7_TYPE;
typedef struct aggcall_type         * AGGCALL_TYPE;
typedef struct comp_type            * COMP_TYPE;
typedef struct quant_type           * QUANT_TYPE;
typedef struct leastmost_type       * LEASTMOST_TYPE;
typedef struct constant_type        * CONSTANT_TYPE;
typedef struct bool_type            * BOOL_TYPE;
typedef struct funcid_type          * FUNCID_TYPE;
typedef struct typeid_type          * TYPEID_TYPE;
typedef struct vblid_type           * VBLID_TYPE;
typedef struct svfuncall_type       * SVFUNCALL_TYPE;
typedef struct mvfuncall_type       * MVFUNCALL_TYPE;
typedef struct singlist_type        * SINGLIST_TYPE;
typedef struct pred_type            * PRED_TYPE;
typedef struct addop_type           * ADDOP_TYPE;
typedef struct mulop_type           * MULOP_TYPE;

/* Declare data structure types     */

typedef union
{
      int   d;
      char  c;
      char  *s;
}UVAL;
```

```
typedef struct identifier_type
{
      UVAL  textval;
      int   type;
}IDENTIFIER_TYPE;

typedef struct identifierv_type
{
      UVAL  textval;
      int   type;
}IDENTIFIERV_TYPE;

typedef struct identifierf_type
{
      UVAL  textval;
      int   type;
}IDENTIFIERF_TYPE;

typedef struct integer_type
{
      UVAL  textval;
      int   type;
}INTEGER_TYPE;

typedef struct string_type
{
      UVAL  textval;
      int   type;
}STRING_TYPE;

typedef struct opr_type
{
      UVAL  textval;
      int   type;
}OPR_TYPE;


/* main data structure declarations      */

STRUCT program_type
{
      int type;
      UNION
      {
            STRUCT branch_program_1
            {
                  STATEMENTS_TYPE         statements_1;
            }BRANCH_program_1;
      }RULE;
};

STRUCT statements_type
{
      int type;
      UNION
      {
            STRUCT branch_statements_1
            {
                  STATEMENT_TYPE          statement_1;
            }BRANCH_statements_1;

            STRUCT branch_statements_2
```

```
            {
                  STRUCT statements_type  *statements_1;
                  STATEMENT_TYPE          statement_2;
            }BRANCH_statements_2;
      }RULE;
};

STRUCT statement_type
{
      int type;
      UNION
      {
            STRUCT branch_statement_1
            {
                  DECLARATIVE_TYPE        declarative_1;
            }BRANCH_statement_1;

            STRUCT branch_statement_2
            {
                  IMPERATIVE_TYPE         imperative_1;
            }BRANCH_statement_2;
      }RULE;
};


STRUCT declarative_type
{
      int type;
      UNION
      {
            STRUCT branch_declarative_1
            {
                  SIMPLE_DECL_TYPE        simple_decl_1;
            }BRANCH_declarative_1;

            STRUCT branch_declarative_2
            {
                  COMPLEX_DECL_TYPE       complex_decl_1;
            }BRANCH_declarative_2;
      }RULE;
};


STRUCT simple_decl_type
{
      int type;
      UNION
      {
            STRUCT branch_simple_decl_1
            {
                  OPR_TYPE                *OPR_1;
                  FUNCID_TYPE             funcid_2;
                  OPR_TYPE                *OPR_3;
                  OPR_TYPE                *OPR_4;
                  ARROW_TYPE              arrow_5;
                  OPR_TYPE                *OPR_6;
            }BRANCH_simple_decl_1;

            STRUCT branch_simple_decl_2
            {
                  OPR_TYPE                *OPR_1;
                  FUNCID_TYPE             funcid_2;
                  OPR_TYPE                *OPR_3;
```

```
                        VBLID_TYPE              vblid_4;
                        OPR_TYPE                *OPR_5;
                        ARROW_TYPE              arrow_6;
                        OPR_TYPE                *OPR_7;
                  }BRANCH_simple_decl_2;

                  STRUCT branch_simple_decl_3
                  {
                        OPR_TYPE                *OPR_1;
                        FUNCID_TYPE             funcid_2;
                        OPR_TYPE                *OPR_3;
                        VBLID_TYPE              vblid_4;
                        OPR_TYPE                *OPR_5;
                        ARROW_TYPE              arrow_6;
                        OPR_TYPE                *OPR_7;
                  }BRANCH_simple_decl_3;
            }RULE;
      };


      STRUCT complex_decl_type
      {
            int type;
            UNION
            {
                  STRUCT branch_complex_decl_1
                  {
                        OPR_TYPE                *OPR_1;
                        FUNCSPEC_TYPE           funcspec_2;
                        ARROW_TYPE              arrow_3;
                        DEFINETYPES_TYPE        definetypes_4;
                  }BRANCH_complex_decl_1;
            }RULE;
      };


      STRUCT definetypes_type
      {
            int type;
            UNION
            {
                  STRUCT branch_definetypes_1
                  {
                        OPR_TYPE                *OPR_1;
                        OPR_TYPE                *OPR_2;
                        FUNCSPEC_TYPE           funcspec_3;
                  }BRANCH_definetypes_1;

                  STRUCT branch_definetypes_2
                  {
                        OPR_TYPE                *OPR_1;
                        OPR_TYPE                *OPR_2;
                        EXPR_TYPE               expr_3;
                  }BRANCH_definetypes_2;

                  STRUCT branch_definetypes_3
                  {
                        OPR_TYPE                *OPR_1;
                        OPR_TYPE                *OPR_2;
                        MTUPLE_TYPE             mtuple_3;
                  }BRANCH_definetypes_3;

                  STRUCT branch_definetypes_4
```

```
            {
                  OPR_TYPE                  *OPR_1;
                  OPR_TYPE                  *OPR_2;
                  MTUPLE_TYPE               mtuple_3;
            }BRANCH_definetypes_4;

            STRUCT branch_definetypes_5
            {
                  OPR_TYPE                  *OPR_1;
                  OPR_TYPE                  *OPR_2;
                  MTUPLE_TYPE               mtuple_3;
            }BRANCH_definetypes_5;

            STRUCT branch_definetypes_6
            {
                  OPR_TYPE                  *OPR_1;
                  OPR_TYPE                  *OPR_2;
                  MTUPLE_TYPE               mtuple_3;
            }BRANCH_definetypes_6;

            STRUCT branch_definetypes_7
            {
                  EXPR_TYPE                 expr_1;
            }BRANCH_definetypes_7;
      }RULE;
};


STRUCT arrow_type
{
      int type;
      UNION
      {
            STRUCT branch_arrow_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_arrow_1;

            STRUCT branch_arrow_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_arrow_2;
      }RULE;
};


STRUCT funcspec_type
{
      int type;
      UNION
      {
            STRUCT branch_funcspec_1
            {
                  FUNCID_TYPE               funcid_1;
                  OPR_TYPE                  *OPR_2;
                  MTUPLE_TYPE               mtuple_3;
                  OPR_TYPE                  *OPR_4;
            }BRANCH_funcspec_1;
      }RULE;
};


STRUCT arglist_type
```

```
{
        int type;
        UNION
        {
                STRUCT branch_arglist_1
                {
                        TYPEID_TYPE             typeid_1;
                }BRANCH_arglist_1;

                STRUCT branch_arglist_2
                {
                        STRUCT arglist_type     *arglist_1;
                        OPR_TYPE                *OPR_2;
                        TYPEID_TYPE             typeid_3;
                }BRANCH_arglist_2;
        }RULE;
};


STRUCT mtuple_type
{
        int type;
        UNION
        {
                STRUCT branch_mtuple_1
                {
                        EXPR_TYPE               expr_1;
                }BRANCH_mtuple_1;

                STRUCT branch_mtuple_2
                {
                        STRUCT mtuple_type      *mtuple_1;
                        OPR_TYPE                *OPR_2;
                        EXPR_TYPE               expr_3;
                }BRANCH_mtuple_2;
        }RULE;
};


STRUCT stuple_type
{
        int type;
        UNION
        {
                STRUCT branch_stuple_1
                {
                        SINGLETON_TYPE          singleton_1;
                }BRANCH_stuple_1;

                STRUCT branch_stuple_2
                {
                        STRUCT stuple_type      *stuple_1;
                        OPR_TYPE                *OPR_2;
                        SINGLETON_TYPE          singleton_3;
                }BRANCH_stuple_2;
        }RULE;
};


STRUCT singlist_type
{
        int type;
        UNION
```

```
        {
                STRUCT branch_singlist_1
                {
                        OPR_TYPE                *OPR_1;
                        SINGLETON_TYPE          singleton_2;
                }BRANCH_singlist_1;

                STRUCT branch_singlist_2
                {
                        STRUCT singlist_type    *singlist_1;
                        OPR_TYPE                *OPR_2;
                        SINGLETON_TYPE          singleton_3;
                }BRANCH_singlist_2;
        }RULE;
};


STRUCT expr_type
{
        int type;
        UNION
        {
                STRUCT branch_expr_1
                {
                        SET_TYPE                set_1;
                }BRANCH_expr_1;

                STRUCT branch_expr_2
                {
                        SINGLETON_TYPE          singleton_1;
                }BRANCH_expr_2;
        }RULE;
};


STRUCT imperative_type
{
        int type;
        UNION
        {
                STRUCT branch_imperative_1
                {
                        FORLOOP_TYPE            forloop_1;
                }BRANCH_imperative_1;

                STRUCT branch_imperative_2
                {
                        GPIMPERATIVE_TYPE       gpimperative_1;
                }BRANCH_imperative_2;
        }RULE;
};


STRUCT forloop_type
{
        int type;
        UNION
        {
                STRUCT branch_forloop_1
                {
                        OPR_TYPE                *OPR_1;
                        OPR_TYPE                *OPR_2;
                        SET_TYPE                set_3;
```

```
                IMPERATIVE_TYPE         imperative_4;
            }BRANCH_forloop_1;

            STRUCT branch_forloop_2
            {
                OPR_TYPE                *OPR_1;
                SINGLETON_TYPE          singleton_2;
                IMPERATIVE_TYPE         imperative_3;
            }BRANCH_forloop_2;
        }RULE;
};


STRUCT gpimperative_type
{
        int type;
        UNION
        {
            STRUCT branch_gpimperative_1
            {
                OPR_TYPE                *OPR_1;
                STUPLE_TYPE             stuple_2;
            }BRANCH_gpimperative_1;
        }RULE;
};


STRUCT set_type
{
        int type;
        UNION
        {
            STRUCT branch_set_1
            {
                TYPEID_TYPE             typeid_1;
            }BRANCH_set_1;

            STRUCT branch_set_2
            {
                MVFUNCALL_TYPE          mvfuncall_1;
            }BRANCH_set_2;

            STRUCT branch_set_3
            {
                OPR_TYPE                *OPR_1;
                STUPLE_TYPE             stuple_2;
                OPR_TYPE                *OPR_3;
            }BRANCH_set_3;

            STRUCT branch_set_4
            {
                OPR_TYPE                *OPR_1;
                SET_TYPE                set_2;
                OPR_TYPE                *OPR_3;
            }BRANCH_set_4;

            STRUCT branch_set_5
            {
                IDENTIFIER_TYPE         *IDENTIFIER_1;
                OPR_TYPE                *OPR_2;
                SET_TYPE                set_3;
            }BRANCH_set_5;
```

```
            STRUCT branch_set_6
            {
                  SET_TYPE                  set_1;
                  OPR_TYPE                  *OPR_2;
                  TYPEID_TYPE               typeid_3;
            }BRANCH_set_6;

            STRUCT branch_set_7
            {
                  SET_TYPE                  set_1;
                  COMP_TYPE                 comp_2;
                  SINGLETON_TYPE            singleton_3;
            }BRANCH_set_7;

            STRUCT branch_set_8
            {
                  SET_TYPE                  set_1;
                  COMP_TYPE                 comp_2;
                  QUANT_TYPE                quant_3;
                  SET_TYPE                  set_4;
            }BRANCH_set_8;

            STRUCT branch_set_9
            {
                  SET_TYPE                  set_1;
                  OPR_TYPE                  *OPR_2;
                  OPR_TYPE                  *OPR_3;
                  PRED_TYPE                 pred_4;
            }BRANCH_set_9;
      }RULE;
};


STRUCT pred_type
{
      int type;
      UNION
      {
            STRUCT branch_pred_1
            {
                  OPR_TYPE                  *OPR_1;
                  SINGLETON_TYPE            singleton_2;
                  PRED_TYPE                 pred_3;
            }BRANCH_pred_1;

            STRUCT branch_pred_2
            {
                  SINGLETON_TYPE            singleton_1;
            }BRANCH_pred_2;

            STRUCT branch_pred_3
            {
                  OPR_TYPE                  *OPR_1;
                  QUANT_TYPE                quant_2;
                  SET_TYPE                  set_3;
                  PRED_TYPE                 pred_4;
            }BRANCH_pred_3;

            STRUCT branch_pred_4
            {
                  SINGLETON_TYPE            singleton_1;
                  COMP_TYPE                 comp_2;
                  SINGLETON_TYPE            singleton_3;
```

```
            }BRANCH_pred_4;

            STRUCT branch_pred_5
            {
                  SINGLETON_TYPE          singleton_1;
                  COMP_TYPE               comp_2;
                  QUANT_TYPE              quant_3;
                  SET_TYPE                set_4;
            }BRANCH_pred_5;

            STRUCT branch_pred_6
            {
                  QUANT_TYPE              quant_1;
                  SET_TYPE                set_2;
                  COMP_TYPE               comp_3;
                  QUANT_TYPE              quant_4;
                  SET_TYPE                set_5;
            }BRANCH_pred_6;
      }RULE;
};


STRUCT singleton_type
{
      int type;
      UNION
      {
            STRUCT branch_singleton_1
            {
                  EXP1_TYPE               exp1_1;
            }BRANCH_singleton_1;

            STRUCT branch_singleton_2
            {
                  EXP1_TYPE               exp1_1;
                  OREXP1_TYPE             orexp1_2;
            }BRANCH_singleton_2;
      }RULE;
};


STRUCT orexp1_type
{
      int type;
      UNION
      {
            STRUCT branch_orexp1_1
            {
                  OPR_TYPE                *OPR_1;
                  EXP1_TYPE               exp1_2;
            }BRANCH_orexp1_1;

            STRUCT branch_orexp1_2
            {
                  STRUCT orexp1_type      *orexp1_1;
                  OPR_TYPE                *OPR_2;
                  EXP1_TYPE               exp1_3;
            }BRANCH_orexp1_2;
      }RULE;
};


STRUCT exp1_type
```

```
{
        int type;
        UNION
        {
                STRUCT branch_exp1_1
                {
                        EXP2_TYPE               exp2_1;
                }BRANCH_exp1_1;

                STRUCT branch_exp1_2
                {
                        EXP2_TYPE               exp2_1;
                        ANDEXP2_TYPE            andexp2_2;
                }BRANCH_exp1_2;
        }RULE;
};


STRUCT andexp2_type
{
        int type;
        UNION
        {
                STRUCT branch_andexp2_1
                {
                        OPR_TYPE                *OPR_1;
                        EXP2_TYPE               exp2_2;
                }BRANCH_andexp2_1;

                STRUCT branch_andexp2_2
                {
                        STRUCT andexp2_type     *andexp2_1;
                        OPR_TYPE                *OPR_2;
                        EXP2_TYPE               exp2_3;
                }BRANCH_andexp2_2;
        }RULE;
};


STRUCT exp2_type
{
        int type;
        UNION
        {
                STRUCT branch_exp2_1
                {
                        EXP3_TYPE               exp3_1;
                }BRANCH_exp2_1;

                STRUCT branch_exp2_2
                {
                        OPR_TYPE                *OPR_1;
                        EXP3_TYPE               exp3_2;
                }BRANCH_exp2_2;
        }RULE;
};


STRUCT exp3_type
{
        int type;
        UNION
        {
```

```
            STRUCT branch_exp3_1
            {
                  EXP4_TYPE                exp4_1;
            }BRANCH_exp3_1;

            STRUCT branch_exp3_2
            {
                  EXP4_TYPE                exp4_1;
                  COMP_TYPE                comp_2;
                  EXP4_TYPE                exp4_3;
            }BRANCH_exp3_2;
      }RULE;
};


STRUCT exp4_type
{
      int type;
      UNION
      {
            STRUCT branch_exp4_1
            {
                  EXP5_TYPE                exp5_1;
            }BRANCH_exp4_1;

            STRUCT branch_exp4_2
            {
                  EXP5_TYPE                exp5_1;
                  ADDOP_TYPE               addop_2;
                  EXP5_TYPE                exp5_3;
            }BRANCH_exp4_2;
      }RULE;
};


STRUCT exp5_type
{
      int type;
      UNION
      {
            STRUCT branch_exp5_1
            {
                  EXP6_TYPE                exp6_1;
            }BRANCH_exp5_1;

            STRUCT branch_exp5_2
            {
                  EXP6_TYPE                exp6_1;
                  MULOP_TYPE               mulop_2;
                  EXP6_TYPE                exp6_3;
            }BRANCH_exp5_2;
      }RULE;
};


STRUCT exp6_type
{
      int type;
      UNION
      {
            STRUCT branch_exp6_1
            {
                  EXP7_TYPE                exp7_1;
```

```
            }BRANCH_exp6_1;

            STRUCT branch_exp6_2
            {
                  EXP7_TYPE                exp7_1;
                  OPR_TYPE                 *OPR_2;
                  TYPEID_TYPE              typeid_3;
            }BRANCH_exp6_2;
      }RULE;
};


STRUCT exp7_type
{
      int type;
      UNION
      {
            STRUCT branch_exp7_1
            {
                  CONSTANT_TYPE            constant_1;
            }BRANCH_exp7_1;

            STRUCT branch_exp7_2
            {
                  VBLID_TYPE               vblid_1;
            }BRANCH_exp7_2;

            STRUCT branch_exp7_3
            {
                  MVFUNCALL_TYPE           mvfuncall_1;
            }BRANCH_exp7_3;

            STRUCT branch_exp7_4
            {
                  AGGCALL_TYPE             aggcall_1;
            }BRANCH_exp7_4;

            STRUCT branch_exp7_5
            {
                  QUANT_TYPE               quant_1;
                  SET_TYPE                 set_2;
                  OPR_TYPE                 *OPR_3;
                  SINGLETON_TYPE           singleton_4;
            }BRANCH_exp7_5;

            STRUCT branch_exp7_6
            {
                  OPR_TYPE                 *OPR_1;
                  SET_TYPE                 set_2;
            }BRANCH_exp7_6;

            STRUCT branch_exp7_7
            {
                  OPR_TYPE                 *OPR_1;
                  SINGLETON_TYPE           singleton_2;
                  OPR_TYPE                 *OPR_3;
            }BRANCH_exp7_7;
      }RULE;
};


STRUCT aggcall_type
{
```

```
int type;
UNION
{
      STRUCT branch_aggcall_1
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SET_TYPE                set_3;
            OPR_TYPE                *OPR_4;
      }BRANCH_aggcall_1;

      STRUCT branch_aggcall_2
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SET_TYPE                set_3;
            OPR_TYPE                *OPR_4;
      }BRANCH_aggcall_2;

      STRUCT branch_aggcall_3
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SET_TYPE                set_3;
            OPR_TYPE                *OPR_4;
      }BRANCH_aggcall_3;

      STRUCT branch_aggcall_4
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SINGLETON_TYPE          singleton_3;
            OPR_TYPE                *OPR_4;
            MTUPLE_TYPE             mtuple_5;
            OPR_TYPE                *OPR_6;
      }BRANCH_aggcall_4;

      STRUCT branch_aggcall_5
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SINGLETON_TYPE          singleton_3;
            OPR_TYPE                *OPR_4;
            MTUPLE_TYPE             mtuple_5;
            OPR_TYPE                *OPR_6;
      }BRANCH_aggcall_5;

      STRUCT branch_aggcall_6
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SINGLETON_TYPE          singleton_3;
            OPR_TYPE                *OPR_4;
      }BRANCH_aggcall_6;

      STRUCT branch_aggcall_7
      {
            OPR_TYPE                *OPR_1;
            OPR_TYPE                *OPR_2;
            SINGLETON_TYPE          singleton_3;
            OPR_TYPE                *OPR_4;
            MTUPLE_TYPE             mtuple_5;
            OPR_TYPE                *OPR_6;
```

```
            }BRANCH_aggcall_7;
      }RULE;
};


STRUCT comp_type
{
      int type;
      UNION
      {
            STRUCT branch_comp_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_1;

            STRUCT branch_comp_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_2;

            STRUCT branch_comp_3
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_3;

            STRUCT branch_comp_4
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_4;

            STRUCT branch_comp_5
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_5;

            STRUCT branch_comp_6
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_comp_6;
      }RULE;
};


STRUCT quant_type
{
      int type;
      UNION
      {
            STRUCT branch_quant_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_quant_1;

            STRUCT branch_quant_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_quant_2;

            STRUCT branch_quant_3
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_quant_3;
```

```
            STRUCT branch_quant_4
            {
                  OPR_TYPE                  *OPR_1;
                  LEASTMOST_TYPE            leastmost_2;
                  SINGLETON_TYPE            singleton_3;
            }BRANCH_quant_4;

            STRUCT branch_quant_5
            {
                  OPR_TYPE                  *OPR_1;
                  SINGLETON_TYPE            singleton_2;
            }BRANCH_quant_5;
      }RULE;
};


STRUCT leastmost_type
{
      int type;
      UNION
      {
            STRUCT branch_leastmost_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_leastmost_1;

            STRUCT branch_leastmost_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_leastmost_2;
      }RULE;
};


STRUCT addop_type
{
      int type;
      UNION
      {
            STRUCT branch_addop_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_addop_1;

            STRUCT branch_addop_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_addop_2;
      }RULE;
};


STRUCT mulop_type
{
      int type;
      UNION
      {
            STRUCT branch_mulop_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_mulop_1;

            STRUCT branch_mulop_2
```

```
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_mulop_2;
      }RULE;
};

STRUCT constant_type
{
      int type;
      UNION
      {
            STRUCT branch_constant_1
            {
                  INTEGER_TYPE              *INTEGER_1;
            }BRANCH_constant_1;

            STRUCT branch_constant_2
            {
                  STRING_TYPE               *STRING_1;
            }BRANCH_constant_2;

            STRUCT branch_constant_3
            {
                  BOOL_TYPE                 bool_1;
            }BRANCH_constant_3;
      }RULE;
};


STRUCT bool_type
{
      int type;
      UNION
      {
            STRUCT branch_bool_1
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_bool_1;

            STRUCT branch_bool_2
            {
                  OPR_TYPE                  *OPR_1;
            }BRANCH_bool_2;
      }RULE;
};


STRUCT funcid_type
{
      int type;
      UNION
      {
            STRUCT branch_funcid_1
            {
                  IDENTIFIERF_TYPE          *IDENTIFIERF_1;
            }BRANCH_funcid_1;
      }RULE;
};


STRUCT typeid_type
{
      int type;
```

```
        UNION
        {
                STRUCT branch_typeid_1
                {
                        IDENTIFIER_TYPE         *IDENTIFIER_1;
                }BRANCH_typeid_1;
        }RULE;
};


STRUCT vblid_type
{
        int type;
        UNION
        {
                STRUCT branch_vblid_1
                {
                        IDENTIFIERV_TYPE        *IDENTIFIERV_1;
                }BRANCH_vblid_1;
        }RULE;
};


STRUCT svfuncall_type
{
        int type;
        UNION
        {
                STRUCT branch_svfuncall_1
                {
                        IDENTIFIERF_TYPE        *IDENTIFIERF_1;
                        OPR_TYPE                *OPR_2;
                        SINGLETON_TYPE          singleton_3;
                        OPR_TYPE                *OPR_4;
                }BRANCH_svfuncall_1;

                STRUCT branch_svfuncall_2
                {
                        IDENTIFIERF_TYPE        *IDENTIFIERF_1;
                        OPR_TYPE                *OPR_2;
                        SINGLETON_TYPE          singleton_3;
                        SINGLIST_TYPE           singlist_4;
                        OPR_TYPE                *OPR_5;
                }BRANCH_svfuncall_2;
        }RULE;
};


STRUCT mvfuncall_type
{
        int type;
        UNION
        {
                STRUCT branch_mvfuncall_1
                {
                        IDENTIFIERF_TYPE        *IDENTIFIERF_1;
                        OPR_TYPE                *OPR_2;
                        MTUPLE_TYPE             mtuple_3;
                        OPR_TYPE                *OPR_4;
                }BRANCH_mvfuncall_1;
        }RULE;
};
```

```
/*    catalog.h file    */


#define MAXLINE                 1000
#define VALIDARGS               2
#define CHILD                   0
#define FAIL                    (-1)
#define PARENT                  default

extern int lineno;

FILE * fp, *dp, *dp2;
char progname[25], dbname[25], incfile[80];
char dbname_rel[50], dbname_func[50], dbname_defs[50];
char syscomm[100];
char rel[] = ".rel";
char func[] = ".func";
char defs[] = ".defs";
int fd;

char *ignore[100];
char endstring[] = "------------|";
char range1[] = "range of a is attribute \n";
char range2[] = "range of r is relation \n";
char retrieve1[] = "retrieve (a.attname) where a.attrelid =    \n";
char retrieve2[] = "retrieve (r.relid) where r.relsave != 0 \n";
char gostate[] = "\\go\n"; /*??escape sequence*/
char quitstate[] = "\\quit\n"; /*??escape sequence*/
char *retstatemnt[100];
char *nullstring;
char *getenv();
char *h;
char *space = " ";
char line[MAXLINE];
char incline[MAXLINE];
int i;
```

```
/*          FILE: trans.h     */


/* includes for this file     */

#include <stdio.h>

/* declarations for this file      */

FILE *fp;
typedef struct RANGELIST
{
      char                    rvar[25];
      char                    range[3];
      struct RANGELIST        *next;
}RANGELIST;

typedef struct VARLIST
{
      char                    var;
      int                     count;
      struct VARLIST          *next;
}VARLIST;

typedef struct FORLIST
{
      int                     type;
      int                     *forptr;
      struct FORLIST          *next;
}FORLIST;

typedef struct DEFLIST
{
      int                     type;
      int                     *defptr;
      struct DEFLIST          *next;
}DEFLIST;

typedef struct PREDLIST
{
      int                     type;
      int                     *predptr;
      struct PREDLIST         *next;
}PREDLIST;

typedef struct CONDLIST
{
      int                     type;
      int                     *condptr;
      struct CONDLIST         *next;
}CONDLIST;

typedef struct LINKLIST
{
      int                     type;
      int                     *linkptr;
      MVFUNCALL_TYPE          mvptr;
      struct LINKLIST         *next;
}LINKLIST;


CONDLIST          *cptr;
PREDLIST          *pptr;
```

```
FORLIST          *fptr;
LINKLIST         *lptr;
DEFLIST          *dptr;
RANGELIST        *rptr;
VARLIST          *vptr;
CONDLIST         condlist;
PREDLIST         predlist;
FORLIST          forlist;
LINKLIST         linklist;
DEFLIST          deflist;
RANGELIST        rangelist;
VARLIST          varlist;
int              phase;
int              found;
```

```
/*          FILE: tabledecls.h      */


int                 relcount, basetot, nbtot;
int                 dertot;
DER_DETAIL          *derptr;
BASE_DETAIL         *bptr;
NB_DETAIL           *nbptr;
DER_DETAIL          der_detail;
BASE_DETAIL         base_detail;
NB_DETAIL           nb_detail;
CATALOG             catalog;
```

```
/*          FILE: tabledefs.h       */


/* This file contains various table definitions      */

/* includes for this file     */

#include "structures.h"

/* locals for this file       */

typedef char                    NAME[25];
typedef NAME                    bfuncname[50];
typedef NAME                    DBNAME;

/*declare table to hold basic functions */

typedef struct BASE_DETAIL
{
      NAME                      funcname;
      struct BASE_DETAIL        *nextfunc;
}BASE_DETAIL;



/*struct for functions which return scalar values or other
entities */

typedef struct NB_DETAIL
{
      NAME                      funcname;         /*function name */
      NAME                      argname;          /*argument */
      NAME                      result_type;      /*result returned*/
      struct NB_DETAIL          *nextfunc;        /*ptr next func*/
}NB_DETAIL;


/*struct for functions which are derived */

typedef struct DER_DETAIL
{
      NAME                      funcname;         /*function name */
      MTUPLE_TYPE               argname;          /*argument */
      DEFINETYPES_TYPE          result_type;      /*return type */
      struct DER_DETAIL         *nextfunc;
}DER_DETAIL;


/*declare table to hold catalog information about a particular
database */

typedef struct ATTRIBUTE
{
      NAME                      attrname;
      NAME                      attrtype;
}ATTRIBUTE;


typedef struct RELATION
{
      NAME                      relname;
      int                       attrcount;
      ATTRIBUTE                 attribute[10];
```

```
}RELATION;

typedef struct CATALOG
{
      DBNAME                  database_name;
      RELATION                relation[10];
}CATALOG;
```

```
/*          FILE: main.c            */

/* This is the main function for the FQLFE system    */

/* Includes for this file     */

#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/dir.h>
#include          <sys/stat.h>
#include          "tabledefs.h"
#include          "tabledecls.h"
#include          "catalog.h"



main(argc, argv)
int argc;
char *argv[];
{

      /* get something onto screen as soon as possible! */
      printf("FQLFE......\n");

      /* check specified database exists */
      if (argc < VALIDARGS)
      {
            printf("FQLFE:Invoke using an existing
                  database\n");
            exit(0);
      }
      strcpy(dbname, argv[1]);

      /* create names of files for this database     */
      sprintf(dbname_rel, "%s%s", dbname, rel);
      sprintf(dbname_func, "%s%s", dbname, func);
      sprintf(dbname_defs, "%s%s", dbname, defs);

      /* initialise totals and pointers to tables    */
      basetot = nbtot = dertot = relcount = 0;
      bptr = &base_detail;
      bptr->nextfunc = NULL;
      nbptr = &nb_detail;
      nbptr->nextfunc = NULL;
      derptr = &der_detail;
      derptr->nextfunc = NULL;

      /* call routine to change directory and check files
            existence   */
      if (check_FQLFE(dbname_rel))
            create_catalog();
      else
            load_catalog();

      /* call routine to change directory and check files
            existence   */
      if (!check_FQLFE(dbname_func))
            create_table();

      /*open the workspace for input     */
      open_workspace("w");
```

```
        /* invoke the monitor eoutine       */
        monitor();

        /* read current totals saved to file by child process      */
        read_totals();

        /* save catalog details to file before system termination */
        save_catalog();
        exit(0);
}



yyerror(s)        /* called for syntax error */
char *s;
{
      printf("FQLFE: Error during syntax analysis\n");
        warning(s, (char *) 0);
}



warning(s, t)    /*print warning message */
char *s, *t;
{
        printf("%s:%s", progname, s);
        if (t)
                printf("%s", t);
        printf("near line %d\n", lineno);
}



monitor()
{
        char word[50];
        char nline[2];
        int one;
        int length, c, inword;

        nline[0] = '\n';
        nline[1] = '\0';
        one = 1;
        length = 0;
        inword = 0;
        system("clear");
        printf("******************WELCOME TO THE FQLFE
                  SYSTEM***************\n");
        printf("\n\n\nready\n");
        printf("**");
        while (getline(line, MAXLINE) > 0)
        {
                i = 0;
                while(line[i] != '\0')
                {
                        if ((line[i] != ' ') && (line[i] != '\n'))
                        {
                                inword = 1;
                                word[length++] = line[i];
                        }
                        else if (inword == 1)
                        {
```

```
                                word[length] = '\0';
                                if(checkword(word, length))
                                    return;
                                else
                                {
                                    length = 0;
                                    inword = 0;
                                }
                        }

                        if (line[i] == '\n')
                        {
                                write_to_file(nline, one);
                                printf("**");
                        }
                        i++;
                }
                if (inword == 1)
                {
                        word[length] = '\0';
                        if(checkword(word, length))
                        return;
                  else
                  {
                                    length = 0;
                                    inword = 0;
                  }
                }

        }
}


getline(s, lim)                 /*get line into s, return length
*/
char s[];
int lim;
{
        int c, i;

        i = 0;
        while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
                s[i++] = c;
        if (c == '\n')
                s[i++] = c;
        s[i] = '\0';
        return(i);
}


checkword(word, length)
char word[];
int length;
{
        int instruction, i;
        static char *commands[] =
        {
                "go",
                "print",
                "include",
                "append",
```

```
                "edit",
                "reset",
                "quit"
        };

        if (word[0] == '\\')
        {
                instruction = 7;
                for (i =0; i < 7; i++)
                {
                        if (strcmp(commands[i],&word[1]) == 0)
                                instruction = i;
            }

                switch (instruction)
                {
                        case 0:
                                        go();
                                        break;
                        case 1:
                                        print();
                                        break;
                        case 2:
                                        include();
                                        break;

                        case 3:
                                        append();
                                        break;
                        case 4:
                                        edit();
                                        break;
                        case 5:
                                        reset();
                                        break;
                        case 6:
                                        return(1);
                                        break;
                        default:
                                        monitor_error(word);
                                        break;
                }

          }
          else
              write_to_file(word, length);
      return(0);
}



go()
{
        int fd, pid, status;
      /*
      extern int yydebug;
      yydebug = 1;
      */
      close_workspace();
      fflush(stdout);
      switch (pid = fork() )
      {
            case FAIL:
```

```
                        system_error("FORK failed\n");
                case CHILD:

                        fflush(stdin);
                        fd = open("FQLFE_workspace", 0);
                        close(0);
                        dup(fd);
                        close(fd);
                        yyparse();
                        fflush(stdout);
                        fflush(stdin);
                        exit(0);


                PARENT:
                        /* in parent */
                        wait(&status);
                        open_workspace("a");

        }
}


print()
{
        int c;
        fclose(dp);
        open_workspace("r");
        while (( c = getc(dp)) != EOF)
                putc(c, stdout);
        close_workspace();
        open_workspace("a");
}


include()
{
      int c;
      /* get name of file to be included */
      get_filename();
      printf("include filename is '%s'\n", incfile);
      close_workspace();
      open_workspace("w");
      if ((dp2 = fopen(incfile, "r")) == NULL)
      {
            FQLFE_error("Unable to open include file\n");
            return;
      }
      /* Copy file incfile to workspace  */
       while ((c = getc(dp2)) != EOF)
      {
            putc(c, dp);
      }
      fclose(dp2);
      close_workspace();
      open_workspace("a");
}


append()
```

```
{
        close_workspace();
        open_workspace("a");
}


edit()
{
        close_workspace();
        system("vi FQLFE_workspace");
        open_workspace("a");
}


reset()
{
        close_workspace();
        open_workspace("w");
}


monitor_error(word)
char word[];
{
        printf("FQLFE: Monitor_error: '%s'\n", word);
}


get_filename()
{
      int j;
      j = 0;
      i++;
      while ((line[i] != ' ') && (line[i] != '\n'))
        {
            incfile[j++] = line[i++];
      }
      incfile[j] = '\0';
}


write_to_file(word, length)
char word[];
int length;
{
        int i;

        for (i = 0; i != length; i++)
        {
                putc(word[i], dp);
        }
        putc(' ', dp);
}


open_workspace(mode)
char mode[];
```

```
{
        if ((dp = fopen("FQLFE_workspace", mode)) == NULL)
        printf("*****FQLFE system error - Unable to open
                workspace*****");
}


close_workspace()
{
        fclose(dp);
}


read_totals()
{

      FILE *dp, *fopen();
      if ((dp = fopen("FQLFE_counts", "r")) == NULL)
      {
            printf("FQLFE:Unable to open file FQLFE_counts\n");
            exit(1);
      }
      fscanf(dp, "%d %d %d", &basetot, &nbtot, &dertot);
      fclose(dp);


}/* end read_totals*/

#include "catalog.c"
```

```
/*          FILE: catalog.c   */


/* These routines generate the catalogue and load up any existing
tables*/




/* This forks a child process which invokes the ingres database
with the given name. It takes its commands from infile and writes
its output to outfile*/

dbexec(infile, outfile)
char *infile, *outfile;
{
      int status, fd, fd2, pid;

      switch (pid = fork() )
        {
          case FAIL:
                        system_error("FORK failed\n");
          case CHILD:

                        /*redirect input */
                        fd = open(infile, 0);
                        close(0);
                        dup(fd);
                        close(fd);

                        /* redirect output */
                        fd2 = open(outfile, 1);
                        close(1);
                        dup(fd2);
                        close(fd2);

                        /* call the Ingres DBMS       */
                        execlp("ingres", "ingres", dbname,
                              (char *) 0);
                        system_error("EXECLP failed\n");

                PARENT:
                        wait(&status);
        }
}


/* This routine uses the result file and extracts the relevant
information about the relations and the attributes */

skiplines(n)
int n;
{
      int i;

      for (i = 0; i < n; i++)
      {
            fgets(ignore, MAXLINE, fp);
      }
}
```

```
/* This creates a table about the database from the system
catalog*/

create_catalog()
{
      /* local variables      */
      int i, k, n, j;
      char name1[25], name2[25], name3[25];
      nullstring = "";

      /* open file to contain catalogue requests     */
      if ((fp = fopen ("catalog_request", "w")) == NULL)
            FQLFE_error("Unable to open file catalog_request");
      /* put out relevant requests */
      sprintf(retrieve2, "help\n");
      fputs(retrieve2, fp);
      fputs(gostate, fp);
      fclose(fp);

      /* call database execution routine */
      dbexec("catalog_request", "result");

      /*open result file */
      if ((fp = fopen ("result", "r")) == NULL)
            FQLFE_error("Unable to open file result");

      /* now scan out relation names for this database*/
      skiplines(12);
      k = 0;
      fscanf(fp, "%s%s%s ", name1, name2, name3);
      while (strcmp(name1, "continue") != 0)
      {
            if (strcmp(name3, "table") == 0)
            {
                  strcpy(catalog.relation[k].relname, name1);
                  k++;
            }
            fscanf(fp, "%s%s%s ", name1, name2, name3);
      }
      relcount = k;
      fclose(fp);

      /*now scan out attribute names for this relation */
      for(i = 0; i < relcount; i++)
      {
            /* open file to contain catalogue requests     */
            if ((fp = fopen("catalog_request", "w")) == NULL)
                  FQLFE_error("Unable to open file
                              catalog_request");

            /* write out requests to file      */
            sprintf(retrieve1, "help %s\n",
                  catalog.relation[i].relname);
            fputs(retrieve1, fp);
            fputs(gostate, fp);
            fclose(fp);

            /* call database execution routine */
            dbexec("catalog_request", "result");

            /*open result file */
```

```
            if ((fp = fopen("result", "r")) == NULL)
                  FQLFE_error("Unable to open file result");

            /* extract relevant information    */
            skiplines(24);
            fscanf(fp, "%s", name1);
            fscanf(fp, "%s", name2);
            fscanf(fp, "%s", ignore);
            fscanf(fp, "%d", ignore);
            j = 0;
            while (strcmp(name1, "Secondary") != 0)
            {
                  strcpy(catalog.relation[i].attribute[j].attrname,
                        name1);
                  if (strcmp(name2, "integer") == 0)
                        strcpy(name2, "INT");
                  else
                        strcpy(name2, "STR");
                  strcpy(catalog.relation[i].attribute[j].attrtype,
                        name2);
                  j++;
                  fscanf(fp, "%s", name1);
                  fscanf(fp, "%s", name2);
                  fscanf(fp, "%s", ignore);
                  fscanf(fp, "%d", ignore);
            }/*end of attribute loop */
            fclose(fp);

            /* store the count of number of attributes for
                  relation    */
            catalog.relation[i].attrcount = j;

      }/*end of relation loop */
}


/* This routine checks to see if the database dbname already has
descriptions for it in the directory $HOME/FAP */

check_FQLFE(name)
char name[];
{

      /* local variables      */
      int fd;
      struct stat stbuf;
      char wd[100], dir[100];

      /* check for environment variable HOME    */
      if (( h = getenv("HOME")) == ((char *) NULL))
      {
            FQLFE_error("No home directory specified\n");
            exit(0);
      }

      /* generate directory name    */
      sprintf(dir, "%s/%s", h, ".FQLFE");

      /* change to the directory $HOME/FAP      */
      if (chdir(dir) == FAIL)
      {
```

```
            /*assume it does not exist and create new one  */
            sprintf(syscomm, "mkdir %s", dir);
            system(syscomm);

            /* attempt to cahnge to the created directory  */
            if (chdir(dir) == FAIL)
            {
                  FQLFE_error("Unable to change to directory
                        $HOME/FAP\n");
                  exit(0);
            }
      }


      /*check file exists     */
        if (stat(name, &stbuf) == FAIL)
      {
                return(1);    /* ie file not there */
      }
        else
      {
                return(0);    /* ie file is there */
      }
}



/* This is an error procesing routine */

FQLFE_error(mess)
char * mess;
{
        printf("FQLFE: Error during automatic function definition
                  phase \n");
        printf("%s\n", mess);
      exit(1);
}



/* This is an error procesing routine for system errors */

system_error(mess)
char * mess;
{
        printf("FQLFE: SYSTEM ERROR\n");
        printf("%s\n", mess);
      exit(1);
}



/* This function reads in the information from the file
$HOME/FAP/dbname and puts it into the table. */

create_table()
{

      /* open the disk file with the table information     */
        if ((fp = fopen(dbname_func, "r")) == NULL)
      {
            printf("FQLFE:Unable to access functional
                  description\n");
```

```
            exit(1);
        }
        /* Keep user informed! */
        printf("Loading up existing functional description of
            database...\n");

        fscanf(fp, "%d %d %d", &basetot, &nbtot, &dertot);

        /* read in the base function table */

        bptr = &base_detail;
        for (i = 0; i < basetot; i++)
        {
            bptr->nextfunc = (BASE_DETAIL *)malloc
                (sizeof(BASE_DETAIL));
            bptr = bptr->nextfunc;
                fscanf(fp, "%s", bptr->funcname);
            bptr->nextfunc = NULL;
        }/* end of loading in base function table */

        /* read in nbfunctable */

        nbptr = &nb_detail;
        for (i = 0; i < nbtot; i++)
        {
            nbptr->nextfunc = (NB_DETAIL *)malloc
                (sizeof(NB_DETAIL));
            nbptr = nbptr->nextfunc;
                fscanf(fp, "%s %s %s", nbptr->funcname, nbptr->
                    argname, nbptr->result_type);
            nbptr->nextfunc = NULL;
          }/*end of loading in nonbase function table */

        derptr = &der_detail;
        for (i = 0; i < dertot; i++)
        {
            derptr->nextfunc = (DER_DETAIL *)malloc
                (sizeof(DER_DETAIL));
            derptr = derptr->nextfunc;
                fscanf(fp, "%s", derptr->funcname);
            derptr->argname = (MTUPLE_TYPE) input_mtuple();
            derptr->result_type = (DEFINETYPES_TYPE)
                input_definetypes();
            derptr->nextfunc = NULL;
        }
}/*end of loading in derived function table */



/* This routine is called when the system fqlfe is exited and
it saves the catalog tables to files. */

save_catalog()
{
        char *filename, *h;
        int j;
        FILE *fp;

        /*open the file to contain the relational description      */
        if ((fp = fopen(dbname_rel, "w")) == NULL)
        {
            printf("FQLFE:Unable to access functional
                description\n");
```

```
        }

        fprintf(fp, "%d%s", relcount, space);
        for (i = 0; i < relcount; i++)
        {
              fprintf(fp, "%d%s", catalog.relation[i].attrcount,
                    space);
        }
        for (i = 0; i < relcount; i++)
        {
              fputs(catalog.relation[i].relname, fp);
              fputs(space, fp);
              for (j = 0; j < catalog.relation[i].attrcount; j++)
              {
                    fputs(catalog.relation[i].attribute[j].attrname,
                          fp);
                    fputs(space, fp);
                    fputs(catalog.relation[i].attribute[j].attrtype,
                          fp);
                    fputs(space, fp);
              }
        }
        fclose(fp);
}/* end of save_catalog */


/* This routine is called when the system fqlfe is started and
it loads the catalog files to tables. */

load_catalog()
{
      char *filename, *h;
      int j;
      FILE *fp;

      /*open the file to contain the relational description      */
      if ((fp = fopen(dbname_rel, "r")) == NULL)
      {
            printf("FQLFE:Unable to access relational
                  description\n");
            exit(1);
      }

      /* Keep user informed! */
      printf("Loading up existing relational description of
            database...\n");

      fscanf(fp, "%d", &relcount);
      for (i = 0; i < relcount; i++)
      {
            fscanf(fp, "%d", &catalog.relation[i].attrcount);
      }
      for (i = 0; i < relcount; i++)
      {
            fscanf(fp, "%s", catalog.relation[i].relname);
            for (j = 0; j < catalog.relation[i].attrcount; j++)
            {
                  fscanf(fp, "%s",
                        catalog.relation[i].attribute[j].attrname);
                  fscanf(fp, "%s",
                        catalog.relation[i].attribute[j].attrtype);
            }
```

```
        }
        fclose(fp);
}/* end of load_catalog */
```

```
/*              FILE: transform.c       */


/* These routines transform the syntax tree of the users input
        */

/* includes for this file      */

#include                "tabledefs.h"
#include                "trans.h"

/* externals for this file     */

extern                  der_lookup();
extern                  check_entity_decl();
extern                  check_str_decl();
extern                  check_int_decl();
extern                  DER_DETAIL der_detail;
extern                  DER_DETAIL *derptr;
extern                  int dertot;
extern                  dbexec();

/* locals for this file        */

VBLID_TYPE              tempvblid;
PRED_TYPE               temppred;
PRED_TYPE               temppred2;
PRED_TYPE               tcondptr;
PRED_TYPE               tcondptr2;
PRED_TYPE               tcondptr3;
char                    *tident;
int                     flag, suchflag,
                        breakflag, predflag,
                        fflag, lflag, gpflag
                        track, identin;



transform_program(tree)
PROGRAM_TYPE    tree;
{
        PROGRAM_TYPE    ptr;
        int             *temp2;
        int             i;


#ifdef  DEBUG
printf("In transform_program\n");
#endif

        flag = suchflag = breakflag = predflag = fflag =  0;
        track = identin = lflag = gpflag = 0;

        ptr = tree;
        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_statements(ptr->
                           RULE.BRANCH_program_1.statements_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
```

```
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_program\n");
#endif

}/* end transform_program*/



transform_statements(ptr)
STATEMENTS_TYPE        ptr;
{

#ifdef  DEBUG
printf("In transform_statements\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_statement(ptr->
                           RULE.BRANCH_statements_1.statement_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_statements(ptr->
                           RULE.BRANCH_statements_2.statements_1);
                        transform_statement(ptr->
                           RULE.BRANCH_statements_2.statement_2);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_statements\n");
#endif

}/* end transform_statements*/



transform_statement(ptr)
STATEMENT_TYPE ptr;
{

#ifdef  DEBUG
printf("In transform_statement\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_declarative(ptr->
                           RULE.BRANCH_statement_1.declarative_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_imperative(ptr->
                           RULE.BRANCH_statement_2.imperative_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
```

```
printf("Out transform_statement\n");
#endif

}/* end transform_statement*/



transform_declarative(ptr)
DECLARATIVE_TYPE        ptr;
{

#ifdef  DEBUG
printf("In transform_declarative\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_simple_decl(ptr->
                           RULE.BRANCH_declarative_1.simple_decl_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_complex_decl(ptr->
                          RULE.BRANCH_declarative_2.complex_decl_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_declarative\n");
#endif

}/* end transform_declarative*/



transform_simple_decl(ptr)
SIMPLE_DECL_TYPE        ptr;
{

#ifdef  DEBUG
printf("In transform_simple_decl\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_simple_decl_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_funcid(ptr->
                                RULE.BRANCH_simple_decl_1.funcid_2);
                        switch(ptr->
                           RULE.BRANCH_simple_decl_1.OPR_3->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
```

```
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            switch(ptr->
               RULE.BRANCH_simple_decl_1.OPR_4->type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            transform_arrow(ptr->
                RULE.BRANCH_simple_decl_1.arrow_5);
            switch(ptr->
               RULE.BRANCH_simple_decl_1.OPR_6_>type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            check_entity_decl;
            break;
case 2:     /*It is alternative  no 2     */
            switch(ptr->
               RULE.BRANCH_simple_decl_2.OPR_1->type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            transform_funcid(ptr->
                RULE.BRANCH_simple_decl_2.funcid_2);
            switch(ptr->
               RULE.BRANCH_simple_decl_2.OPR_3->type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            transform_vblid(ptr->
               RULE.BRANCH_simple_decl_2.vblid_4);
            switch(ptr->
               RULE.BRANCH_simple_decl_2.OPR_5->type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            transform_arrow(ptr->
               RULE.BRANCH_simple_decl_2.arrow_6);
            switch(ptr->
               RULE.BRANCH_simple_decl_2.OPR_7->type)
            {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
            }/* end switch */
            check_int_decl;
```

```
                        break;
                case 3:         /*It is alternative  no 3      */
                        switch(ptr->
                           RULE.BRANCH_simple_decl_3.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_funcid(ptr->
                           RULE.BRANCH_simple_decl_3.funcid_2);
                        switch(ptr->
                           RULE.BRANCH_simple_decl_3.OPR_3->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_vblid(ptr->
                           RULE.BRANCH_simple_decl_3.vblid_4);
                        switch(ptr->
                           RULE.BRANCH_simple_decl_3.OPR_5->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_arrow(ptr->
                           RULE.BRANCH_simple_decl_3.arrow_6);
                        switch(ptr->
                           RULE.BRANCH_simple_decl_3.OPR_7->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        check_str_decl;
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_simple_decl\n");
#endif

}/* end transform_simple_decl*/



transform_complex_decl(ptr)
COMPLEX_DECL_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_complex_decl\n");
#endif

        switch(ptr->type)
        {
```

```
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_complex_decl_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_funcspec(ptr->
                           RULE.BRANCH_complex_decl_1.funcspec_2);
                        transform_arrow(ptr->
                           RULE.BRANCH_complex_decl_1.arrow_3);
                        transform_definetypes(ptr->
                           RULE.BRANCH_complex_decl_1.definetype_4);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_complex_decl\n");
#endif

}/* end transform_complex_decl*/



transform_definetypes(ptr)
DEFINETYPES_TYPE        ptr;
{

#ifdef  DEBUG
printf("In transform_definetypes\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_definetypes_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        switch(ptr->
                           RULE.BRANCH_definetypes_1.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_funcspec(ptr->
                           RULE.BRANCH_definetypes_1.funcspec_3);
                        check_inverse();
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->
                           RULE.BRANCH_definetypes_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
```

```
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        switch(ptr->
           RULE.BRANCH_definetypes_2.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                type generated\n");
        }/* end switch */
        transform_expr(ptr->
           RULE.BRANCH_definetypes_2.expr_3);
        check_trans();
        break;
case 3:         /*It is alternative  no 3      */
        switch(ptr->
           RULE.BRANCH_definetypes_3.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        switch(ptr->
           RULE.BRANCH_definetypes_3.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        transform_mtuple(ptr->
           RULE.BRANCH_definetypes_3.mtuple_3);
        check_comp();
        break;
case 4:         /*It is alternative  no 4      */
        switch(ptr->
           RULE.BRANCH_definetypes_4.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        switch(ptr->
           RULE.BRANCH_definetypes_4.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                type generated\n");
        }/* end switch */
        transform_mtuple(ptr->
           RULE.BRANCH_definetypes_4.mtuple_3);
        check_inter();
        break;
case 5:         /*It is alternative  no 5      */
        switch(ptr->
           RULE.BRANCH_definetypes_5.OPR_1->type)
        {
                case 1:         /*It is an operator*/
```

```
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                                }/* end switch */
                                switch(ptr->
                                   RULE.BRANCH_definetypes_5.OPR_2->type)
                                {
                                        case 1:         /*It is an operator*/
                                                break;
                                        default:printf("ERROR : Wrong lexeme
                                                type generated\n");
                                }/* end switch */
                                transform_mtuple(ptr->
                                   RULE.BRANCH_definetypes_5.mtuple_3);
                                check_union();
                                break;
                        case 6:         /*It is alternative  no 6      */
                                switch(ptr->
                                   RULE.BRANCH_definetypes_6.OPR_1->type)
                                {
                                        case 1:         /*It is an operator*/
                                                break;
                                        default:printf("ERROR : Wrong lexeme
                                                type generated\n");
                                }/* end switch */
                                switch(ptr->
                                   RULE.BRANCH_definetypes_6.OPR_2->type)
                                {
                                        case 1:         /*It is an operator*/
                                                break;
                                        default:printf("ERROR : Wrong lexeme
                                                type generated\n");
                                }/* end switch */
                                transform_mtuple(ptr->
                                   RULE.BRANCH_definetypes_6.mtuple_3);
                                check_diff();
                                break;
                        case 7:         /*It is alternative  no 7      */
                                transform_expr(ptr->
                                   RULE.BRANCH_definetypes_7.expr_1);
                                check_expr();
                                break;
                        default:printf("ERROR: illegal alternative no.\n");
                }/* end switch */

#ifdef  DEBUG
printf("Out transform_definetypes\n");
#endif

}/* end transform_definetypes*/



transform_arrow(ptr)
ARROW_TYPE      ptr;
{

#ifdef  DEBUG
printf("In transform_arrow\n");
#endif

        switch(ptr->type)
        {
```

```
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->RULE.BRANCH_arrow_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->RULE.BRANCH_arrow_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_arrow\n");
#endif

}/* end transform_arrow*/



transform_funcspec(ptr)
FUNCSPEC_TYPE   ptr;
{

#ifdef  DEBUG
printf("In transform_funcspec\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_funcid(ptr->
                           RULE.BRANCH_funcspec_1.funcid_1);
                        switch(ptr->
                           RULE.BRANCH_funcspec_1.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_mtuple(ptr->
                           RULE.BRANCH_funcspec_1.mtuple_3);
                        switch(ptr->
                           RULE.BRANCH_funcspec_1.OPR_4->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */
```

```
#ifdef  DEBUG
printf("Out transform_funcspec\n");
#endif

}/* end transform_funcspec*/



transform_arglist(ptr)
ARGLIST_TYPE    ptr;
{

#ifdef  DEBUG
printf("In transform_arglist\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_typeid(ptr->
                           RULE.BRANCH_arglist_1.typeid_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_arglist(ptr->
                           RULE.BRANCH_arglist_2.arglist_1);
                        switch(ptr->
                           RULE.BRANCH_arglist_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_typeid(ptr->
                           RULE.BRANCH_arglist_2.typeid_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_arglist\n");
#endif

}/* end transform_arglist*/



transform_mtuple(ptr)
MTUPLE_TYPE     ptr;
{

#ifdef  DEBUG
printf("In transform_mtuple\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_expr(ptr->
                           RULE.BRANCH_mtuple_1.expr_1);
                        break;
                case 2:         /*It is alternative  no 2      */
```

```
                        transform_mtuple(ptr->
                           RULE.BRANCH_mtuple_2.mtuple_1);
                        switch(ptr->
                           RULE.BRANCH_mtuple_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(", ", stdout);
                                fputs(", ", fp);
                        }

                        transform_expr(ptr->
                           RULE.BRANCH_mtuple_2.expr_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_mtuple\n");
#endif

}/* end transform_mtuple*/



transform_stuple(ptr)
STUPLE_TYPE     ptr;
{

#ifdef  DEBUG
printf("In transform_stuple\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_singleton(ptr->
                           RULE.BRANCH_stuple_1.singleton_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_stuple(ptr->
                           RULE.BRANCH_stuple_2.stuple_1);
                        switch(ptr->
                           RULE.BRANCH_stuple_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(", ", stdout);
                                fputs(", ", fp);
                        }
                        transform_singleton(ptr->
                           RULE.BRANCH_stuple_2.singleton_3);
                        break;
```

```
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_stuple\n");
#endif

}/* end transform_stuple*/



transform_singlist(ptr)
SINGLIST_TYPE  ptr;
{

#ifdef  DEBUG
printf("In transform_singlist\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_singlist_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_singlist_1.singleton_2);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_singlist(ptr->
                           RULE.BRANCH_singlist_2.singlist_1);
                        switch(ptr->
                           RULE.BRANCH_singlist_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_singlist_2.singleton_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_singlist\n");
#endif

}/* end transform_singlist*/



transform_expr(ptr)
EXPR_TYPE       ptr;
{

#ifdef  DEBUG
```

```
printf("In transform_expr\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_set(ptr->
                           RULE.BRANCH_expr_1.set_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_singleton(ptr->
                           RULE.BRANCH_expr_2.singleton_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_expr\n");
#endif

}/* end transform_expr*/



transform_imperative(ptr)
IMPERATIVE_TYPE         ptr;
{

#ifdef  DEBUG
printf("In transform_imperative\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        /*phase 1 initialisations */
                        if (phase ==  1)
                        {
                        rptr = &rangelist;
                        rptr->next = NULL;
                        vptr = &varlist;
                        vptr->next = NULL;
                        fptr = &forlist;
                        fptr->next = NULL;
                        lptr = &linklist;
                        lptr->next = NULL;
                        pptr = &predlist;
                        pptr->next = NULL;
                        cptr = &condlist;
                        cptr->next = NULL;
                        dptr = &deflist;
                        dptr->next = NULL;
                        }

                        /*code for phase 2 transformations */

                        transform_forloop(ptr->
                           RULE.BRANCH_imperative_1.forloop_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        /*code for phase 2 transformations */
                        if (phase == 2)
                        {
```

```
                        cptr->next = (CONDLIST *)malloc
                                (sizeof(CONDLIST));
                        cptr = cptr->next;
                        cptr->condptr = (int *)ptr;
                        cptr->type = 1;
                        cptr->next = NULL;
                        }
                        transform_gpimperative(ptr->
                           RULE.BRANCH_imperative_2.gpimperative_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_imperative\n");
#endif

}/* end transform_imperative*/



transform_forloop(ptr)
FORLOOP_TYPE    ptr;
{

#ifdef  DEBUG
printf("In transform_forloop\n");
#endif

        fflag = 1;
        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        /*code for phase 2 transformations */
                        if (phase == 2)
                        {
                        printf("ENTRY into FORLIST in forloop\n");
                        fptr->next = (FORLIST *)malloc
                           (sizeof(FORLIST));
                        fptr = fptr->next;
                        fptr->forptr = (int *)ptr;
                        fptr->type = 1;
                        fptr->next = NULL;
                        }
                        switch(ptr->
                           RULE.BRANCH_forloop_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        switch(ptr->
                          RULE.BRANCH_forloop_1.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs("\nrange of ", stdout);
```

```
                                fputs("\nrange of ", fp);
                        }
                        transform_set(ptr->
                           RULE.BRANCH_forloop_1.set_3);
                        if (phase == 3)
                                break;
                        transform_imperative(ptr->
                           RULE.BRANCH_forloop_1.imperative_4);
                        break;
                case 2:         /*It is alternative  no 2      */
                        /*code for phase 2 transformations */
                        if (phase == 2)
                        {
                        printf("ENTRY into FORLIST in forloop\n");
                        fptr->next = (FORLIST *)malloc
                                (sizeof(FORLIST));
                        fptr = fptr->next;
                        fptr->forptr = (int *)ptr;
                        fptr->type = 1;
                        fptr->next = NULL;
                        }
                        switch(ptr->
                           RULE.BRANCH_forloop_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_forloop_2.singleton_2);
                        if (phase == 3)
                                break;
                        transform_imperative(ptr->
                           RULE.BRANCH_forloop_2.imperative_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_forloop\n");
#endif

}/* end transform_forloop*/



transform_gpimperative(ptr)
GPIMPERATIVE_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_gpimperative\n");
#endif

        gpflag = 1;
        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_gpimperative_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
```

```
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs("\nretrieve ", stdout);
                                fputs("\nretrieve ", fp);
                                fputs(" ( ", stdout);
                                fputs(" ( ", fp);
                        }
                        transform_stuple(ptr->
                           RULE.BRANCH_gpimperative_1.stuple_2);
                        if (phase == 3)
                        {
                                fputs(" ) ", stdout);
                                fputs(" ) ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_gpimperative\n");
#endif

}/* end transform_gpimperative*/



transform_set(ptr)
SET_TYPE        ptr;
{

#ifdef  DEBUG
printf("In transform_set\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_typeid(ptr->
                           RULE.BRANCH_set_1.typeid_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_mvfuncall(ptr->
                           RULE.BRANCH_set_2.mvfuncall_1);
                        break;
                case 3:         /*It is alternative  no 3      */
                        switch(ptr->RULE.BRANCH_set_3.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ( ", stdout);
                                fputs(" ( ", fp);
                        }
                        transform_stuple(ptr->
                           RULE.BRANCH_set_3.stuple_2);
```

```
        switch(ptr->RULE.BRANCH_set_3.OPR_3->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" ) ", stdout);
                fputs(" ) ", fp);
        }
        break;
case 4:         /*It is alternative  no 4      */
        switch(ptr->RULE.BRANCH_set_4.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        transform_set(ptr->RULE.BRANCH_set_4.set_2);
        switch(ptr->RULE.BRANCH_set_4.OPR_3->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        break;
case 5:         /*It is alternative  no 5      */
        switch(ptr->
           RULE.BRANCH_set_5.IDENTIFIER_1->type)
        {
                case 2:         /*It is an ident*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_set_5.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                type generated\n");
        }/* end switch */
        identin = 1;
        tident = ptr->
           RULE.BRANCH_set_5.IDENTIFIER_1->
                textval.s;
        transform_set(ptr->RULE.BRANCH_set_5.set_3);
        identin = 0;
        break;
case 6:         /*It is alternative  no 6      */
        transform_set(ptr->RULE.BRANCH_set_6.set_1);
        switch(ptr->RULE.BRANCH_set_6.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
```

```
                        transform_typeid(ptr->
                           RULE.BRANCH_set_6.typeid_3);
                        break;
                case 7:         /*It is alternative  no 7      */
                        transform_set(ptr->RULE.BRANCH_set_7.set_1);
                        transform_comp(ptr->
                           RULE.BRANCH_set_7.comp_2);
                        transform_singleton(ptr->
                           RULE.BRANCH_set_7.singleton_3);
                        break;
                case 8:         /*It is alternative  no 8      */
                        transform_set(ptr->RULE.BRANCH_set_8.set_1);
                        transform_comp(ptr->
                           RULE.BRANCH_set_8.comp_2);
                        transform_quant(ptr->
                           RULE.BRANCH_set_8.quant_3);
                        transform_set(ptr->RULE.BRANCH_set_8.set_4);
                        break;
                case 9:         /*It is alternative  no 9      */
                        transform_set(ptr->RULE.BRANCH_set_9.set_1);
                        switch(ptr->RULE.BRANCH_set_9.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        switch(ptr->RULE.BRANCH_set_9.OPR_3->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        if (breakflag == 1)
                        {
                                tcondptr = ptr->
                                        RULE.BRANCH_set_9.pred_4;
                                break;
                        }
                        suchflag = 1;
                        if (fflag == 1 && suchflag == 1 &&
                                        gpflag == 0)
                                predflag = 1;
                        if (phase == 3 && suchflag == 1)
                        {
                                fputs(" where ", stdout);
                                fputs(" where ", fp);
                        }
                        transform_pred(ptr->
                           RULE.BRANCH_set_9.pred_4);
                        suchflag = 0;
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_set\n");
#endif

}/* end transform_set*/
```

```
transform_pred(ptr)
PRED_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_pred\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        /*code for phase 2 transformations */
                        if (phase == 2)
                        {
                        printf("ENTRY into FORLIST in pred\n");
                        fptr->next = (FORLIST *)malloc
                                (sizeof(FORLIST));
                        fptr = fptr->next;
                        fptr->forptr = (int *)ptr;
                        fptr->type = 2;
                        fptr->next = NULL;
                        }
                        switch(ptr->RULE.BRANCH_pred_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_pred_1.singleton_2);
                        if (phase == 3)
                        if (breakflag == 1)
                        {
                                tcondptr = ptr->
                           RULE.BRANCH_pred_1.pred_3;
                                break;
                        }
                        transform_pred(ptr->
                           RULE.BRANCH_pred_1.pred_3);
                        break;
                case 2:         /*It is alternative  no 2      */
                        /*code for phase 2 transformations */
                        transform_singleton(ptr->
                           RULE.BRANCH_pred_2.singleton_1);
                        break;
                case 3:         /*It is alternative  no 3      */
                        /*code for phase 2 transformations */
                        if (phase == 2)
                        {
                        printf("ENTRY into FORLIST in pred\n");
                        fptr->next = (FORLIST *)malloc
                                (sizeof(FORLIST));
                        fptr = fptr->next;
                        fptr->forptr = (int *)ptr;
                        fptr->type = 2;
                        fptr->next = NULL;
                        }
                        switch(ptr->RULE.BRANCH_pred_3.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
```

```
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_quant(ptr->
                           RULE.BRANCH_pred_3.quant_2);
                        if (phase == 3)
                        {
                                fputs("\nrange of ", stdout);
                                fputs("\nrange of ", fp);
                        }
                        flag = 1;
                        transform_set(ptr->
                           RULE.BRANCH_pred_3.set_3);
                        flag = 0;
                        if (phase == 3)
                        if (breakflag == 1)
                        {
                                tcondptr = ptr->
                           RULE.BRANCH_pred_3.pred_4;
                                break;
                        }
                        transform_pred(ptr->
                           RULE.BRANCH_pred_3.pred_4);
                        break;
                case 4:         /*It is alternative  no 4      */
                        transform_singleton(ptr->
                           RULE.BRANCH_pred_4.singleton_1);
                        transform_comp(ptr->
                           RULE.BRANCH_pred_4.comp_2);
                        transform_singleton(ptr->
                           RULE.BRANCH_pred_4.singleton_3);
                        break;
                case 5:         /*It is alternative  no 5      */
                        transform_singleton(ptr->
                           RULE.BRANCH_pred_5.singleton_1);
                        transform_comp(ptr->
                           RULE.BRANCH_pred_5.comp_2);
                        transform_quant(ptr->
                           RULE.BRANCH_pred_5.quant_3);
                        transform_set(ptr->
                           RULE.BRANCH_pred_5.set_4);
                        break;
                case 6:         /*It is alternative  no 6      */
                        transform_quant(ptr->
                           RULE.BRANCH_pred_6.quant_1);
                        transform_set(ptr->
                           RULE.BRANCH_pred_6.set_2);
                        transform_comp(ptr->
                           RULE.BRANCH_pred_6.comp_3);
                        transform_quant(ptr->
                           RULE.BRANCH_pred_6.quant_4);
                        transform_set(ptr->
                           RULE.BRANCH_pred_6.set_5);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_pred\n");
#endif

}/* end transform_pred*/
```

```
transform_singleton(ptr)
SINGLETON_TYPE ptr;
{

#ifdef  DEBUG
printf("In transform_singleton\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_expl(ptr->
                           RULE.BRANCH_singleton_1.expl_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_expl(ptr->
                           RULE.BRANCH_singleton_2.expl_1);
                        transform_orexpl(ptr->
                           RULE.BRANCH_singleton_2.orexpl_2);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_singleton\n");
#endif

}/* end transform_singleton*/



transform_orexpl(ptr)
OREXPl_TYPE     ptr;
{

#ifdef  DEBUG
printf("In transform_orexpl\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_orexpl_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" or ", stdout);
                                fputs(" or ", fp);
                        }
                        transform_expl(ptr->
                           RULE.BRANCH_orexpl_1.expl_2);
                        break;
                case 2:         /*It is alternative  no 2      */
```

```
                        transform_orexp1(ptr->
                           RULE.BRANCH_orexp1_2.orexp1_1);
                        switch(ptr->
                           RULE.BRANCH_orexp1_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" or ", stdout);
                                fputs(" or ", fp);
                        }
                        transform_exp1(ptr->
                           RULE.BRANCH_orexp1_2.exp1_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_orexp1\n");
#endif

}/* end transform_orexp1*/



transform_exp1(ptr)
EXP1_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_exp1\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_exp2(ptr->
                           RULE.BRANCH_exp1_1.exp2_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_exp2(ptr->
                           RULE.BRANCH_exp1_2.exp2_1);
                        transform_andexp2(ptr->
                           RULE.BRANCH_exp1_2.andexp2_2);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp1\n");
#endif

}/* end transform_exp1*/



transform_andexp2(ptr)
ANDEXP2_TYPE    ptr;
{
```

```
#ifdef  DEBUG
printf("In transform_andexp2\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_andexp2_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" and ", stdout);
                                fputs(" and ", fp);
                        }
                        transform_exp2(ptr->
                           RULE.BRANCH_andexp2_1.exp2_2);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_andexp2(ptr->
                           RULE.BRANCH_andexp2_2.andexp2_1);
                        switch(ptr->
                           RULE.BRANCH_andexp2_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" and ", stdout);
                                fputs(" and ", fp);
                        }
                        transform_exp2(ptr->
                           RULE.BRANCH_andexp2_2.exp2_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_andexp2\n");
#endif

}/* end transform_andexp2*/

transform_exp2(ptr)
EXP2_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_exp2\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
```

```
                        transform_exp3(ptr->
                           RULE.BRANCH_exp2_1.exp3_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->RULE.BRANCH_exp2_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_exp3(ptr->
                           RULE.BRANCH_exp2_2.exp3_2);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp2\n");
#endif

}/* end transform_exp2*/



transform_exp3(ptr)
EXP3_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_exp3\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_exp4(ptr->
                           RULE.BRANCH_exp3_1.exp4_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_exp4(ptr->
                           RULE.BRANCH_exp3_2.exp4_1);
                        transform_comp(ptr->
                           RULE.BRANCH_exp3_2.comp_2);
                        transform_exp4(ptr->
                           RULE.BRANCH_exp3_2.exp4_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp3\n");
#endif

}/* end transform_exp3*/



transform_exp4(ptr)
EXP4_TYPE       ptr;
{

#ifdef  DEBUG
```

```
printf("In transform_exp4\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_exp5(ptr->
                           RULE.BRANCH_exp4_1.exp5_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_exp5(ptr->
                           RULE.BRANCH_exp4_2.exp5_1);
                        transform_addop(ptr->
                           RULE.BRANCH_exp4_2.addop_2);
                        transform_exp5(ptr->
                           RULE.BRANCH_exp4_2.exp5_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp4\n");
#endif

}/* end transform_exp4*/



transform_exp5(ptr)
EXP5_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_exp5\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_exp6(ptr->
                           RULE.BRANCH_exp5_1.exp6_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_exp6(ptr->
                           RULE.BRANCH_exp5_2.exp6_1);
                        transform_mulop(ptr->
                           RULE.BRANCH_exp5_2.mulop_2);
                        transform_exp6(ptr->
                           RULE.BRANCH_exp5_2.exp6_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp5\n");
#endif

}/* end transform_exp5*/



transform_exp6(ptr)
EXP6_TYPE       ptr;
```

```
{

#ifdef  DEBUG
printf("In transform_exp6\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_exp7(ptr->
                           RULE.BRANCH_exp6_1.exp7_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_exp7(ptr->
                           RULE.BRANCH_exp6_2.exp7_1);
                        switch(ptr->RULE.BRANCH_exp6_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_typeid(ptr->
                           RULE.BRANCH_exp6_2.typeid_3);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp6\n");
#endif

}/* end transform_exp6*/



transform_exp7(ptr)
EXP7_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_exp7\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        transform_constant(ptr->
                           RULE.BRANCH_exp7_1.constant_1);
                        break;
                case 2:         /*It is alternative  no 2      */
                        transform_vblid(ptr->
                           RULE.BRANCH_exp7_2.vblid_1);
                        break;
                case 3:         /*It is alternative  no 3      */
                        transform_mvfuncall(ptr->
                           RULE.BRANCH_exp7_3.mvfuncall_1);
                        break;
                case 4:         /*It is alternative  no 4      */
                        transform_aggcall(ptr->
                           RULE.BRANCH_exp7_4.aggcall_1);
                        break;
                case 5:         /*It is alternative  no 5      */
```

```
                        transform_quant(ptr->
                           RULE.BRANCH_exp7_5.quant_1);
                        transform_set(ptr->
                           RULE.BRANCH_exp7_5.set_2);
                        switch(ptr->RULE.BRANCH_exp7_5.OPR_3->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_exp7_5.singleton_4);
                        break;
                case 6:         /*It is alternative  no 6      */
                        switch(ptr->RULE.BRANCH_exp7_6.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_set(ptr->
                            RULE.BRANCH_exp7_6.set_2);
                        break;
                case 7:         /*It is alternative  no 7      */
                        switch(ptr->RULE.BRANCH_exp7_7.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ( ", stdout);
                                fputs(" ( ", fp);
                        }
                        transform_singleton(ptr->
                           RULE.BRANCH_exp7_7.singleton_2);
                        switch(ptr->RULE.BRANCH_exp7_7.OPR_3->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ) ", stdout);
                                fputs(" ) ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_exp7\n");
#endif

}/* end transform_exp7*/
```

```
transform_aggcall(ptr)
AGGCALL_TYPE   ptr;
{

#ifdef  DEBUG
printf("In transform_aggcall\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_aggcall_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" count ", stdout);
                                fputs(" count ", fp);
                        }
                        switch(ptr->
                           RULE.BRANCH_aggcall_1.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ( ", stdout);
                                fputs(" ( ", fp);
                        }
                        transform_set(ptr->
                           RULE.BRANCH_aggcall_1.set_3);
                        switch(ptr->
                           RULE.BRANCH_aggcall_1.OPR_4->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ) ", stdout);
                                fputs(" ) ", fp);
                        }
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->
                           RULE.BRANCH_aggcall_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
```

```
                if (phase == 3)
                {
                        fputs(" max ", stdout);
                        fputs(" max ", fp);
                }
                switch(ptr->
                   RULE.BRANCH_aggcall_2.OPR_2->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" ( ", stdout);
                        fputs(" ( ", fp);
                }
                transform_set(ptr->
                   RULE.BRANCH_aggcall_2.set_3);
                switch(ptr->
                   RULE.BRANCH_aggcall_2.OPR_4->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" ) ", stdout);
                        fputs(" ) ", fp);
                }
                break;
        case 3:         /*It is alternative  no 3      */
                switch(ptr->
                   RULE.BRANCH_aggcall_3.OPR_1->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" min ", stdout);
                        fputs(" min ", fp);
                }
                switch(ptr->
                   RULE.BRANCH_aggcall_3.OPR_2->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" ( ", stdout);
                        fputs(" ( ", fp);
                }
                transform_set(ptr->
                   RULE.BRANCH_aggcall_3.set_3);
```

```
                switch(ptr->
                   RULE.BRANCH_aggcall_3.OPR_4->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" ) ", stdout);
                        fputs(" ) ", fp);
                }
                break;
        case 4:         /*It is alternative  no 4      */
                switch(ptr->
                   RULE.BRANCH_aggcall_4.OPR_1->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" sum ", stdout);
                        fputs(" sum ", fp);
                }
                switch(ptr->
                   RULE.BRANCH_aggcall_4.OPR_2->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" ( ", stdout);
                        fputs(" ( ", fp);
                }
                transform_singleton(ptr->
                   RULE.BRANCH_aggcall_4.singleton_3);
                switch(ptr->
                   RULE.BRANCH_aggcall_4.OPR_4->type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs(" by ", stdout);
                        fputs(" by ", fp);
                }
                transform_mtuple(ptr->
                   RULE.BRANCH_aggcall_4.mtuple_5);
                switch(ptr->
                   RULE.BRANCH_aggcall_4.OPR_6->type)
                {
                        case 1:         /*It is an operator*/
                                break;
```

```
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" ) ", stdout);
                fputs(" ) ", fp);
        }
        break;
case 5:         /*It is alternative  no 5     */
        switch(ptr->
           RULE.BRANCH_aggcall_5.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong
                        lexeme type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" avg ", stdout);
                fputs(" avg ", fp);
        }
        switch(ptr->
           RULE.BRANCH_aggcall_5.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" ( ", stdout);
                fputs(" ( ", fp);
        }
        transform_singleton(ptr->
           RULE.BRANCH_aggcall_5.singleton_3);
        switch(ptr->
           RULE.BRANCH_aggcall_5.OPR_4->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" by ", stdout);
                fputs(" by ", fp);
        }
        transform_mtuple(ptr->
           RULE.BRANCH_aggcall_5.mtuple_5);
        switch(ptr->
           RULE.BRANCH_aggcall_5.OPR_6->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
```

```
                fputs(" ) ", stdout);
                fputs(" ) ", fp);
        }
        break;
case 6:         /*It is alternative  no 6      */
        switch(ptr->
           RULE.BRANCH_aggcall_6.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" avg ", stdout);
                fputs(" avg ", fp);
        }
        switch(ptr->
           RULE.BRANCH_aggcall_6.OPR_2->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" ( ", stdout);
                fputs(" ( ", fp);
        }
        transform_singleton(ptr->
           RULE.BRANCH_aggcall_6.singleton_3);
        switch(ptr->
           RULE.BRANCH_aggcall_6.OPR_4->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" ) ", stdout);
                fputs(" ) ", fp);
        }
        break;
case 7:         /*It is alternative  no 7      */
        switch(ptr->
           RULE.BRANCH_aggcall_7.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" count ", stdout);
                fputs(" count ", fp);
        }
        switch(ptr->
           RULE.BRANCH_aggcall_7.OPR_2->type)
```

```
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ( ", stdout);
                                fputs(" ( ", fp);
                        }
                        transform_singleton(ptr->
                           RULE.BRANCH_aggcall_7.singleton_3);
                        switch(ptr->
                           RULE.BRANCH_aggcall_7.OPR_4->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" by ", stdout);
                                fputs(" by ", fp);
                        }
                        transform_mtuple(ptr->
                           RULE.BRANCH_aggcall_7.mtuple_5);
                        switch(ptr->
                           RULE.BRANCH_aggcall_7.OPR_6->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" ) ", stdout);
                                fputs(" ) ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_aggcall\n");
#endif

}/* end transform_aggcall*/



transform_comp(ptr)
COMP_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_comp\n");
#endif

        switch(ptr->type)
        {
```

```
case 1:         /*It is alternative  no 1      */
        switch(ptr->RULE.BRANCH_comp_1.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" > ", stdout);
                fputs(" > ", fp);
        }
        break;
case 2:         /*It is alternative  no 2      */
        switch(ptr->RULE.BRANCH_comp_2.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" < ", stdout);
                fputs(" < ", fp);
        }
        break;
case 3:         /*It is alternative  no 3      */
        switch(ptr->RULE.BRANCH_comp_3.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" = ", stdout);
                fputs(" = ", fp);
        }
        break;
case 4:         /*It is alternative  no 4      */
        switch(ptr->RULE.BRANCH_comp_4.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
        }/* end switch */
        if (phase == 3)
        {
                fputs(" != ", stdout);
                fputs(" != ", fp);
        }
        break;
case 5:         /*It is alternative  no 5      */
        switch(ptr->RULE.BRANCH_comp_5.OPR_1->type)
        {
                case 1:         /*It is an operator*/
                        break;
                default:printf("ERROR : Wrong lexeme
                        type generated\n");
```

```
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" <= ", stdout);
                                fputs(" <= ", fp);
                        }
                        break;
                case 6:         /*It is alternative  no 6     */
                        switch(ptr->RULE.BRANCH_comp_6.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs(" >= ", stdout);
                                fputs(" >= ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_comp\n");
#endif

}/* end transform_comp*/



transform_quant(ptr)
QUANT_TYPE      ptr;
{

#ifdef  DEBUG
printf("In transform_quant\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1     */
                        switch(ptr->RULE.BRANCH_quant_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                case 2:         /*It is alternative  no 2     */
                        switch(ptr->RULE.BRANCH_quant_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                case 3:         /*It is alternative  no 3     */
                        switch(ptr->RULE.BRANCH_quant_3.OPR_1->type)
                        {
```

```
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                case 4:         /*It is alternative  no 4      */
                        switch(ptr->RULE.BRANCH_quant_4.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_leastmost(ptr->
                           RULE.BRANCH_quant_4.leastmost_2);
                        transform_singleton(ptr->
                           RULE.BRANCH_quant_4.singleton_3);
                        break;
                case 5:         /*It is alternative  no 5      */
                        switch(ptr->RULE.BRANCH_quant_5.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_quant_5.singleton_2);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_quant\n");
#endif

}/* end transform_quant*/



transform_leastmost(ptr)
LEASTMOST_TYPE ptr;
{

#ifdef  DEBUG
printf("In transform_leastmost\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_leastmost_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->
```

```
                        RULE.BRANCH_leastmost_2.OPR_1->type)
                    {
                        case 1:         /*It is an operator*/
                            break;
                        default:printf("ERROR : Wrong lexeme
                            type generated\n");
                    }/* end switch */
                    break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_leastmost\n");
#endif

}/* end transform_leastmost*/



transform_addop(ptr)
ADDOP_TYPE      ptr;
{

#ifdef  DEBUG
printf("In transform_addop\n");
#endif

        switch(ptr->type)
        {
            case 1:         /*It is alternative  no 1      */
                    switch(ptr->RULE.BRANCH_addop_1.OPR_1->type)
                    {
                        case 1:         /*It is an operator*/
                            break;
                        default:printf("ERROR : Wrong lexeme
                            type generated\n");
                    }/* end switch */
                    if (phase == 3)
                        fputs(" + ",fp);
                    break;
            case 2:         /*It is alternative  no 2      */
                    switch(ptr->RULE.BRANCH_addop_2.OPR_1->type)
                    {
                        case 1:         /*It is an operator*/
                            break;
                        default:printf("ERROR : Wrong lexeme
                            type generated\n");
                    }/* end switch */
                    if (phase == 3)
                        fputs(" - ",fp);
                    break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_addop\n");
#endif

}/* end transform_addop*/



transform_mulop(ptr)
```

```
MULOP_TYPE      ptr;
{

#ifdef  DEBUG
printf("In transform_mulop\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->RULE.BRANCH_mulop_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                                fputs(" * ",fp);
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->RULE.BRANCH_mulop_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                                fputs(" / ",fp);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_mulop\n");
#endif

}/* end transform_mulop*/



transform_constant(ptr)
CONSTANT_TYPE  ptr;
{

#ifdef  DEBUG
printf("In transform_constant\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_constant_1.INTEGER_1->type)
                        {
                                case 3:         /*It is an integer*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
```

```
                                fprintf(stdout, " %d ", (ptr->
                                RULE.BRANCH_constant_1.INTEGER_1->
                                        textval.d));
                                fprintf(fp, " %d ", (ptr->
                                RULE.BRANCH_constant_1.INTEGER_1->
                                        textval.d));
                        }
                        break;
                case 2:         /*It is alternative  no 2     */
                        switch(ptr->
                           RULE.BRANCH_constant_2.STRING_1->type)
                        {
                                case 4:         /*It is a string*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs((ptr->
                                RULE.BRANCH_constant_2.STRING_1->
                                        textval.s), stdout);
                                fputs((ptr->
                                RULE.BRANCH_constant_2.STRING_1->
                                        textval.s), fp);
                        }
                        break;
                case 3:         /*It is alternative  no 3     */
                        transform_bool(ptr->
                                RULE.BRANCH_constant_3.bool_1);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_constant\n");
#endif

}/* end transform_constant*/



transform_bool(ptr)
BOOL_TYPE       ptr;
{

#ifdef  DEBUG
printf("In transform_bool\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1     */
                        switch(ptr->RULE.BRANCH_bool_1.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
```

```
                                fputs((ptr->
                                RULE.BRANCH_bool_1.OPR_1->
                                        textval.s),fp);
                        break;
                case 2:         /*It is alternative  no 2     */
                        switch(ptr->RULE.BRANCH_bool_2.OPR_1->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                                fputs((ptr->
                                RULE.BRANCH_bool_2.OPR_1->
                                        textval.s),fp);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_bool\n");
#endif

}/* end transform_bool*/



transform_funcid(ptr)
FUNCID_TYPE     ptr;
{
        int i;

#ifdef  DEBUG
printf("In transform_funcid\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1     */
                        switch(ptr->
                           RULE.BRANCH_funcid_1.IDENTIFIERF_1->type)
                        {
                                case 6:         /*It is an ident*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        /*phase 1 transformations */
                        if (phase == 3)
                        {
                                fputs((ptr->
                                RULE.BRANCH_funcid_1.IDENTIFIERF_1
                                        ->textval.s), stdout);
                                fputs((ptr->
                                RULE.BRANCH_funcid_1.IDENTIFIERF_1
                                        ->textval.s), fp);
                                fputs(" ", stdout);
                                fputs(" ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */
```

```
#ifdef  DEBUG
printf("Out transform_funcid\n");
#endif

}/* end transform_funcid*/



transform_typeid(ptr)
TYPEID_TYPE     ptr;
{
        int i;

#ifdef  DEBUG
printf("In transform_typeid\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_typeid_1.IDENTIFIER_1->type)
                        {
                                case 2:         /*It is an ident*/
                                        break;
                                default:
                                        if (phase != 3)
                                                printf("ERROR : Wrong
                                        lexeme type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                rptr = &rangelist;
                                while (rptr->next != NULL)
                                        rptr = rptr->next;
                                rptr->next = (RANGELIST *)malloc
                                        (sizeof(RANGELIST));
                                rptr = rptr->next;
                                rptr->next = NULL;
                                strcpy(rptr->rvar,
                                (ptr->
                                RULE.BRANCH_typeid_1.IDENTIFIER_1->
                                        textval.s));
                                i = 0;
                                rptr->range[i] = rptr->rvar[i];
                                i++;
                                vptr = &varlist;
                                found = 0;
                                while(vptr->next != NULL)
                                {
                                        vptr = vptr->next;
                                        if (vptr->var == rptr->
                                                range[0])
                                        {
                                                found = 1;
                                                break;
                                        }
                                }
                                if (found == 0)
                                {
                                        vptr->next = (VARLIST *)
                                           malloc(sizeof(VARLIST));
```

```
                                        vptr = vptr->next;
                                        vptr->var = rptr->range[0];
                                        vptr->count = 0;
                                        vptr->next = NULL;
                                }
                                vptr->count = (vptr->count) + 1;
                                if ((vptr->count) > 1)
                                {
                                        rptr->range[i] =((vptr->
                                           count) - 1) %10 + '0';
                                        i++;
                                }
                                rptr->range[i] = '\0';
                                if (identin == 1)
                                        strcpy(rptr->range, tident);
                                fputs(rptr->range, stdout);
                                fputs(rptr->range, fp);
                                fputs(" is ", stdout);
                                fputs(" is ", fp);
                                fputs(rptr->rvar, stdout);
                                fputs(rptr->rvar, fp);
                        }
                        break;
                default:
                        if (phase != 3)
                                printf("ERROR: illegal alternative
                                no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_typeid\n");
#endif

}/* end transform_typeid*/



transform_vblid(ptr)
VBLID_TYPE      ptr;
{

#ifdef  DEBUG
printf("In transform_vblid\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_vblid_1.IDENTIFIERV_1->type)
                        {
                                case 5:         /*It is an var
                                                identifier      */
                                        break;
                                default:
                                        if (phase != 3)
                                                printf("ERROR : Wrong
                                           lexeme type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                /*check range values in case vblid
                                        is a range var */
```

```
                                found = 0;
                                rptr = &rangelist;
                                while (rptr->next != NULL)
                                {
                                        rptr = rptr->next;
                                        if (strcmp(rptr->range,
                                                ptr->
                                                RULE.BRANCH_vblid_1.
                                                IDENTIFIERV_1->
                                                textval.s) == 0)
                                        {
                                                found = 1;
                                                fputs(rptr->range,
                                                        stdout);
                                                fputs(rptr->range,
                                                        fp);
                                                fputs(".", stdout);
                                                fputs(".", fp);
                                                break;
                                        }
                                }
                                if (found == 1)
                                        break;
                                rptr = &rangelist;
                                while (rptr->next != NULL)
                                {
                                        rptr = rptr->next;
                                        if (strcmp(rptr->rvar,
                                        ptr->
                                                RULE.BRANCH_vblid_1.
                                                IDENTIFIERV_1
                                                ->textval.s) == 0)
                                        {
                                                found = 1;
                                                break;
                                        }
                                }
                                fputs(rptr->range, stdout);
                                fputs(rptr->range, fp);
                                fputs(".", stdout);
                                fputs(".", fp);
                        }
                        break;
                default:
                        if (phase != 3)
                                printf("ERROR: illegal alternative
                                        no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_vblid\n");
#endif

}/* end transform_vblid*/



transform_svfuncall(ptr)
SVFUNCALL_TYPE ptr;
{

#ifdef  DEBUG
printf("In transform_svfuncall\n");
```

```
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_svfuncall_1.IDENTIFIERF_1->
                                type)
                        {
                                case 6:         /*It is an ident*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */

                        switch(ptr->
                           RULE.BRANCH_svfuncall_1.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        transform_singleton(ptr->
                           RULE.BRANCH_svfuncall_1.singleton_3);
                        switch(ptr->
                           RULE.BRANCH_svfuncall_1.OPR_4->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs((ptr->
                                RULE.BRANCH_svfuncall_1.
                                IDENTIFIERF_1->textval.s), stdout);
                                fputs((ptr->
                                RULE.BRANCH_svfuncall_1.
                                IDENTIFIERF_1->textval.s), fp);
                                fputs(" ", stdout);
                                fputs(" ", fp);
                        }
                        break;
                case 2:         /*It is alternative  no 2      */
                        switch(ptr->
                           RULE.BRANCH_svfuncall_2.IDENTIFIERF_1->
                                type)
                        {
                                case 6:         /*It is an ident*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        switch(ptr->
                           RULE.BRANCH_svfuncall_2.OPR_2->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
```

```
                        transform_singleton(ptr->
                           RULE.BRANCH_svfuncall_2.singleton_3);
                        transform_singlist(ptr->
                           RULE.BRANCH_svfuncall_2.singlist_4);
                        switch(ptr->
                           RULE.BRANCH_svfuncall_2.OPR_5->type)
                        {
                                case 1:         /*It is an operator*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        if (phase == 3)
                        {
                                fputs((ptr->
                                RULE.BRANCH_svfuncall_2.
                                IDENTIFIERF_1->textval.s), stdout);
                                fputs((ptr->
                                RULE.BRANCH_svfuncall_2.
                                IDENTIFIERF_1->textval.s), fp);
                                fputs(" ", stdout);
                                fputs(" ", fp);
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_svfuncall\n");
#endif

}/* end transform_svfuncall*/



transform_mvfuncall(ptr)
MVFUNCALL_TYPE ptr;
{

#ifdef  DEBUG
printf("In transform_mvfuncall\n");
#endif

        switch(ptr->type)
        {
                case 1:         /*It is alternative  no 1      */
                        switch(ptr->
                           RULE.BRANCH_mvfuncall_1.IDENTIFIERF_1->
                                type)
                        {
                                case 6:         /*It is an ident*/
                                        break;
                                default:printf("ERROR : Wrong lexeme
                                        type generated\n");
                        }/* end switch */
                        /*phase 1 transformations */
                        found = 0;
                        if (phase == 1)
                        der_lookup(ptr->
                           RULE.BRANCH_mvfuncall_1.
                                IDENTIFIERF_1->textval.s,
                                ptr->
                                  RULE.BRANCH_mvfuncall_1.mtuple_3);
```

```
if (found)
        if (flag == 1)
        {
        ptr->type = 3;
        traverse_definetypes(derptr->
                result_type);
        (VBLID_TYPE) ptr->
                RULE.BRANCH_mvfuncall_1.
                IDENTIFIERF_1->textval.s
                        =  tempvblid;
        tempvblid->type = 1;
        while (temppred != NULL)
        {
                if (temppred->type != 1 &&
                        temppred->type != 3)
                {
                temppred2 = temppred;
                temppred = NULL;
                }
                else
                {
                dptr->next = (DEFLIST *)
                   malloc(sizeof(DEFLIST));
                dptr = dptr->next;
                dptr->defptr = (int *)
                        temppred;
                dptr->type = 3;
                dptr->next = NULL;
                temppred2 = temppred;
                temppred = NULL;
                if (temppred2->type == 1)

                traverse_pred(temppred2->
                RULE.BRANCH_pred_1.pred_3);
                else

                traverse_pred(temppred2->
                RULE.BRANCH_pred_3.pred_4);
                }
        }
        lptr->next = (LINKLIST *)malloc
                (sizeof(LINKLIST));
        lptr = lptr->next;
        lptr->linkptr = (int *)temppred2;
        lptr->mvptr = ptr;
        lptr->type = 2;
        lptr->next = NULL;
        }
        else if (suchflag == 1)
        {
        ptr->type = 4;
        dptr->next = (DEFLIST *)malloc
                (sizeof(DEFLIST));
        dptr = dptr->next;
        dptr->defptr = (int *)derptr->
                result_type;
        dptr->type = 1;
        dptr->next = NULL;

        lptr->next = (LINKLIST *)malloc
                (sizeof(LINKLIST));
        lptr = lptr->next;
```

```
                lptr->linkptr = (int *)derptr->
                        result_type;
                lptr->mvptr = ptr;
                lptr->type = 1;
                lptr->next = NULL;
                }
                else
                {
                traverse_definetypes(derptr->
                        result_type);
                (VBLID_TYPE) ptr->
                RULE.BRANCH_mvfuncall_1.
                IDENTIFIERF_1->textval.s =
                        tempvblid;
                tempvblid->type = 1;
                ptr->type = 2;
                dptr->next = (DEFLIST *)malloc
                        (sizeof(DEFLIST));
                dptr = dptr->next;
                dptr->defptr = (int *)derptr->
                        result_type;
                dptr->type = 4;
                dptr->next = NULL;
                while (temppred != NULL)
                {
                        if (temppred->type != 1 &&
                                temppred->type != 3)
                        {
                        temppred2 = temppred;
                        temppred = NULL;
                        }
                        else
                        {
                        dptr->next = (DEFLIST *)
                           malloc(sizeof(DEFLIST));
                        dptr = dptr->next;
                        dptr->defptr = (int *)
                                temppred;
                        dptr->type = 3;
                        dptr->next = NULL;
                        temppred2 = temppred;
                        temppred = NULL;
                        if (temppred2->type == 1)

                        traverse_pred(temppred2->
                        RULE.BRANCH_pred_1.pred_3);
                        else

                        traverse_pred(temppred2->
                        RULE.BRANCH_pred_3.pred_4);
                        }
                }
                lptr->next = (LINKLIST *)malloc
                        (sizeof(LINKLIST));
                lptr = lptr->next;
                lptr->linkptr = (int *)temppred2;
                lptr->mvptr = ptr;
                lptr->type = 1;
                lptr->next = NULL;
                }
        else
        {
        track = suchflag;
```

```
                suchflag = 0;
                switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_2->
                        type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                transform_mtuple(ptr->
                   RULE.BRANCH_mvfuncall_1.mtuple_3);
                switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_4->
                        type)
                {
                        case 1:         /*It is an operator*/
                                break;
                        default:printf("ERROR : Wrong lexeme
                                type generated\n");
                }/* end switch */
                if (phase == 3)
                {
                        fputs((ptr->
                           RULE.BRANCH_mvfuncall_1.
                                IDENTIFIERF_1->
                                textval.s), stdout);
                        fputs((ptr->
                           RULE.BRANCH_mvfuncall_1.
                                IDENTIFIERF_1->
                                textval.s), fp);
                        fputs(" ", stdout);
                        fputs(" ", fp);
                }
                }/*end else */
                break;
        case 2:         /*It is alternative  no 2 -only
                        entered in phase 2 & 3*/
                /*phase 2 transformations */
                if (phase == 3)
                {
                /* look up linklist for this funcs address
                   if there transform the rhs of the
                   definetype from pred onwards ie ignore
                   set SUCH THAT */
                        if (link_lookup(ptr) == 1)
                        {
                                /*
                                if (track == 1)
                                {
                                fputs(" where ", stdout);
                                fputs(" where ", fp);
                                }
                                */

                                transform_pred((PRED_TYPE)
                                        lptr->linkptr);
                        }
                        else
                                printf("LINK ERROR\n");
                        fputs(" and ", stdout);
                        fputs(" and ", fp);
                        transform_vblid((VBLID_TYPE) ptr->
                           RULE.BRANCH_mvfuncall_1.
                                IDENTIFIERF_1->textval.s);
```

```
                        }
                        break;
                case 3:/*It is alt no 3 */
                        /*phase 2 transformations */
                        if (phase == 3)
                        {
                        /* look up linklist for this funcs address
                           if there transform the rhs of the
                           definetype from pred onwards ie ignore
                           set SUCH THAT */
                                transform_typeid((TYPEID_TYPE)ptr->
                                   RULE.BRANCH_mvfuncall_1.
                                        IDENTIFIERF_1->textval.s);
                                if (link_lookup(ptr) == 1)
                                {
                                        tcondptr3 = (PRED_TYPE)
                                                lptr->linkptr;
                                        lflag = 1;
                                }
                                else
                                        printf("LINK ERROR\n");
                        }
                        break;
                case 4:         /*It is alternative  no 2 -only
                                entered in phase 2 & 3*/
                        /*phase 2 transformations */
                        if (phase == 3)
                        {
                        /* look up linklist for this funcs address
                           if there transform the rhs of the
                           definetype from pred onwards ie ignore
                           set SUCH THAT */
                                if (link_lookup(ptr) == 1)
                                {
                                        traverse_definetypes
                                                (lptr->linkptr);
                                        transform_pred(temppred);
                                }
                                else
                                        printf("LINK ERROR\n");
                        }
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef  DEBUG
printf("Out transform_mvfuncall\n");
#endif

}/* end transform_mvfuncall*/



phase_three()
{
        int first;

#ifdef  DEBUG
printf("In phase_three\n");
#endif

        /*still in phase 2!!*/
        /*transform DEFLIST */
```

```
dptr = &deflist;
for (; dptr->next != NULL; )
{
        dptr = dptr->next;
        switch(dptr->type)
        {
                case 1:

                        transform_definetypes(
                           (DEFINETYPES_TYPE)dptr->defptr);
                        break;
                case 2:
                        break;
                case 3:
                        break;
                case 4:
                        break;
                default:printf("ERROR: illegal alternative
                        no.\n");
        }/* end switch */
}

if ((fp = fopen("FQLFE_query", "w")) == NULL)
        printf("Unable to open FQLFE_query\n");
printf("\nTransformed query is.......\n");
phase = 3;
breakflag = 1;

/*transform DEFLIST */
dptr = &deflist;
for (; dptr->next != NULL; )
{
        dptr = dptr->next;
        switch(dptr->type)
        {
                case 1:
                        fputs(" range of ", stdout);
                        fputs(" range of ", fp);

                        transform_definetypes(
                           (DEFINETYPES_TYPE)dptr->defptr);
                        break;
                case 2:
                        fputs(" range of ", stdout);
                        fputs(" range of ", fp);
                        transform_pred(
                           (PRED_TYPE)dptr->defptr);
                        break;
                case 3:
                        transform_pred(
                           (PRED_TYPE)dptr->defptr);
                        break;
                case 4:
                        fputs(" range of ", stdout);
                        fputs(" range of ", fp);
                        transform_definetypes(
                           (DEFINETYPES_TYPE)dptr->defptr);
                        break;
                default:printf("ERROR: illegal alternative
                        no.\n");
        }/* end switch */
}
```

```
/*transform FORLIST */
fptr = &forlist;
for (; fptr->next != NULL; )
{
        fptr = fptr->next;
        switch(fptr->type)
        {
                case 1:

                transform_forloop(
                        (FORLOOP_TYPE)fptr->forptr);
                        break;
                case 2:
                        transform_pred(
                           (PRED_TYPE)fptr->forptr);
                        break;
                default:printf("ERROR: illegal alternative
                        no.\n");
        }/* end switch */
}
breakflag = 0;
tcondptr2 = tcondptr;

/*transform CONDLIST */
cptr = &condlist;
for (; cptr->next != NULL; )
{
        cptr = cptr->next;
        switch(cptr->type)
        {
                case 1:

                        transform_imperative(
                           (IMPERATIVE_TYPE)cptr->condptr);
                        break;
                default:printf("ERROR: illegal alternative
                        no.\n");
        }/* end switch */
}

/*transform PREDLIST */
if (predflag == 1)
{
        fputs(" where ", stdout);
        fputs(" where ", fp);
        transform_pred(tcondptr2);
        if (tcondptr2 != tcondptr3)
                if (lflag == 1)
                {
                        fputs(" and ", stdout);
                        fputs(" and ", fp);
                        transform_pred(tcondptr3);
                }
}
fputs("\n\\go\n", fp);
fputs("\\quit\n", fp);
fclose(fp);
/* invoke dbexec with FQLFE_query as input and FQLFE_query
        as output */
printf("\nexecuting transformed query......\n");
dbexec("FQLFE_query", "result");
printf("\nresult......\n");
system("cat result");
```

```
#ifdef  DEBUG
printf("Out phase_three\n");
#endif

}/* end phase_three*/



link_lookup(ptr)
MVFUNCALL_TYPE ptr;
{

#ifdef  DEBUG
printf("In link_lookup\n");
#endif

        lptr = &linklist;
        while (lptr->next != NULL)
        {
                lptr = lptr->next;
                if (lptr->mvptr == ptr)
                        return(1);
        }

#ifdef  DEBUG
printf("Out link_lookup\n");
#endif
        return(0);
}
```

```
/*          FILE: traverse.c  */


/* These routines traverse the syntax tree of the users input    */

/* includes for this file     */

#include          <stdio.h>
#include          "tabledefs.h"

/* externals for this file    */

extern            check_entity_decl();
extern            check_int_decl();
extern            check_str_decl();
extern            phase;
extern            VBLID_TYPE        tempvblid;
extern            PRED_TYPE         temppred;
extern            int               basetot, nbtot, dertot;
extern            DER_DETAIL        *derptr;
extern            BASE_DETAIL       *bptr;
extern            NB_DETAIL         *nbptr;
extern            DER_DETAIL        der_detail;
extern            BASE_DETAIL       base_detail;
extern            NB_DETAIL         nb_detail;
extern            save_table();

/*    locals for this file    */

int indef = 0;
int linenum = 0;




/*Routine to walk a program branch of the syntax tree*/

traverse_program(tree)
PROGRAM_TYPE      tree;
{
      PROGRAM_TYPE      ptr;
      int               *temp2;

#ifdef      DEBUG
printf("In traverse_program\n");
#endif

      ptr = tree;
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_statements(ptr->
                        RULE.BRANCH_program_1.statements_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

/*    output_program(ptr);    */
      write_totals();
      save_table();

#ifdef      DEBUG
```

```
printf("Out traverse_program\n");
#endif

}/* end traverse_program*/



/*Routine to walk a statement branch of the syntax tree*/

traverse_statements(ptr)
STATEMENTS_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_statements\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_statement(ptr->
                        RULE.BRANCH_statements_1.statement_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_statements(ptr->
                        RULE.BRANCH_statements_2.statements_1);
                  traverse_statement(ptr->
                        RULE.BRANCH_statements_2.statement_2);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_statements\n");
#endif

}/* end traverse_statements*/



/*Routine to walk a statement branch of the syntax tree*/

traverse_statement(ptr)
STATEMENT_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_statement\n");
#endif

      linenum++;
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_declarative(ptr->
                        RULE.BRANCH_statement_1.declarative_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_imperative(ptr->
                        RULE.BRANCH_statement_2.imperative_1);
                  if (indef == 0)
                  {
                  phase = 1;
```

```
                    transform_imperative(ptr->
                          RULE.BRANCH_statement_2.imperative_1);
                    phase = 2;
                    transform_imperative(ptr->
                          RULE.BRANCH_statement_2.imperative_1);
                    phase_three();
                    }
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_statement\n");
#endif

}/* end traverse_statement*/



/*Routine to walk a declarative branch of the syntax tree*/

traverse_declarative(ptr)
DECLARATIVE_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_declarative\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_simple_decl(ptr->
                        RULE.BRANCH_declarative_1.simple_decl_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_complex_decl(ptr->
                        RULE.BRANCH_declarative_2.complex_decl_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_declarative\n");
#endif

}/* end traverse_declarative*/



/*Routine to walk a simple_decl branch of the syntax tree*/

traverse_simple_decl(ptr)
SIMPLE_DECL_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_simple_decl\n");
#endif

      indef = 1;
      switch(ptr->type)
      {
```

```
case 1:     /*It is alternative  no 1     */
      switch(ptr->RULE.BRANCH_simple_decl_1.OPR_1->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      traverse_funcid(ptr->
            RULE.BRANCH_simple_decl_1.funcid_2);
      switch(ptr->RULE.BRANCH_simple_decl_1.OPR_3->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      switch(ptr->RULE.BRANCH_simple_decl_1.OPR_4->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      traverse_arrow(ptr->
            RULE.BRANCH_simple_decl_1.arrow_5);
      switch(ptr->RULE.BRANCH_simple_decl_1.OPR_6->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      if (phase == 0)
      check_entity_decl(ptr);
      break;
case 2:     /*It is alternative  no 2     */
      switch(ptr->RULE.BRANCH_simple_decl_2.OPR_1->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      traverse_funcid(ptr->
            RULE.BRANCH_simple_decl_2.funcid_2);
      switch(ptr->RULE.BRANCH_simple_decl_2.OPR_3->
            type)
      {
            case 1:     /*It is an operator     */
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      traverse_vblid(ptr->
            RULE.BRANCH_simple_decl_2.vblid_4);
      switch(ptr->RULE.BRANCH_simple_decl_2.OPR_5->
            type)
```

```
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_arrow(ptr->
                        RULE.BRANCH_simple_decl_2.arrow_6);
                switch(ptr->RULE.BRANCH_simple_decl_2.OPR_7->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                if (phase == 0)
                check_int_decl(ptr);
                break;
        case 3:     /*It is alternative  no 3     */
                switch(ptr->RULE.BRANCH_simple_decl_3.OPR_1->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_funcid(ptr->
                        RULE.BRANCH_simple_decl_3.funcid_2);
                switch(ptr->RULE.BRANCH_simple_decl_3.OPR_3->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_vblid(ptr->
                        RULE.BRANCH_simple_decl_3.vblid_4);
                switch(ptr->RULE.BRANCH_simple_decl_3.OPR_5->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_arrow(ptr->
                        RULE.BRANCH_simple_decl_3.arrow_6);
                switch(ptr->RULE.BRANCH_simple_decl_3.OPR_7->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                if (phase == 0)
                check_str_decl(ptr);
                break;
        default:printf("ERROR: illegal alternative no.\n");
}/* end switch */
```

```
        indef = 0;

#ifdef      DEBUG
printf("Out traverse_simple_decl\n");
#endif

}/* end traverse_simple_decl*/



/*Routine to walk a complex_decl branch of the syntax tree*/

traverse_complex_decl(ptr)
COMPLEX_DECL_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_complex_decl\n");
#endif

        indef = 1;
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    switch(ptr->RULE.BRANCH_complex_decl_1.OPR_1->
                          type)
                    {
                          case 1:     /*It is an operator     */
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    traverse_funcspec(ptr->
                          RULE.BRANCH_complex_decl_1.funcspec_2);
                    traverse_arrow(ptr->
                          RULE.BRANCH_complex_decl_1.arrow_3);
                    traverse_definetypes(ptr->
                          RULE.BRANCH_complex_decl_1.definetypes_4);
                    if (phase == 0)
                    check_comp_decl(ptr);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */
        indef = 0;

#ifdef      DEBUG
printf("Out traverse_complex_decl\n");
#endif

}/* end traverse_complex_decl*/



/*Routine to walk a definetypes branch of the syntax tree*/

traverse_definetypes(ptr)
DEFINETYPES_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_definetypes\n");
#endif
```

```
temppred = NULL;
tempvblid = NULL;
switch(ptr->type)
{
      case 1:     /*It is alternative  no 1     */
            switch(ptr->RULE.BRANCH_definetypes_1.OPR_1->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_definetypes_1.OPR_2->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            traverse_funcspec(ptr->
                  RULE.BRANCH_definetypes_1.funcspec_3);
            if (phase == 0)
            check_inverse();
            break;
      case 2:     /*It is alternative  no 2     */
            switch(ptr->RULE.BRANCH_definetypes_2.OPR_1->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_definetypes_2.OPR_2->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            traverse_expr(ptr->
                  RULE.BRANCH_definetypes_2.expr_3);
            if (phase == 0)
            check_trans();
            break;
      case 3:     /*It is alternative  no 3     */
            switch(ptr->RULE.BRANCH_definetypes_3.OPR_1->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_definetypes_3.OPR_2->
                  type)
            {
                  case 1:     /*It is an operator     */
                        break;
```

```
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                traverse_mtuple(ptr->
                        RULE.BRANCH_definetypes_3.mtuple_3);
                if (phase == 0)
                check_comp();
                break;
        case 4:     /*It is alternative  no 4     */
                switch(ptr->RULE.BRANCH_definetypes_4.OPR_1->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                switch(ptr->RULE.BRANCH_definetypes_4.OPR_2->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_mtuple(ptr->
                        RULE.BRANCH_definetypes_4.mtuple_3);
                if (phase == 0)
                check_inter();
                break;
        case 5:     /*It is alternative  no 5     */
                switch(ptr->RULE.BRANCH_definetypes_5.OPR_1->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                switch(ptr->RULE.BRANCH_definetypes_5.OPR_2->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_mtuple(ptr->
                        RULE.BRANCH_definetypes_5.mtuple_3);
                if (phase == 0)
                check_union();
                break;
        case 6:     /*It is alternative  no 6     */
                switch(ptr->RULE.BRANCH_definetypes_6.OPR_1->
                        type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                switch(ptr->RULE.BRANCH_definetypes_6.OPR_2->
                        type)
```

```
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_mtuple(ptr->
                        RULE.BRANCH_definetypes_6.mtuple_3);
                if (phase == 0)
                check_diff();
                break;
        case 7:     /*It is alternative  no 7     */
                traverse_expr(ptr->
                        RULE.BRANCH_definetypes_7.expr_1);
                if (phase == 0)
                check_expr();
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_definetypes\n");
#endif

}/* end traverse_definetypes*/



/*Routine to walk a arrow branch of the syntax tree*/

traverse_arrow(ptr)
ARROW_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_arrow\n");
#endif

    switch(ptr->type)
    {
        case 1:     /*It is alternative  no 1     */
                switch(ptr->RULE.BRANCH_arrow_1.OPR_1->type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        case 2:     /*It is alternative  no 2     */
                switch(ptr->RULE.BRANCH_arrow_2.OPR_1->type)
                {
                        case 1:     /*It is an operator     */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_arrow\n");
```

```
#endif

}/* end traverse_arrow*/



/*Routine to walk a funcspec branch of the syntax tree*/

traverse_funcspec(ptr)
FUNCSPEC_TYPE     ptr;
{

#ifdef      DEBUG
printf("In traverse_funcspec\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_funcid(ptr->
                        RULE.BRANCH_funcspec_1.funcid_1);
                  switch(ptr->RULE.BRANCH_funcspec_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_mtuple(ptr->
                        RULE.BRANCH_funcspec_1.mtuple_3);
                  switch(ptr->RULE.BRANCH_funcspec_1.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_funcspec\n");
#endif

}/* end traverse_funcspec*/



/*Routine to walk an arglist branch of the syntax tree */

traverse_arglist(ptr)
ARGLIST_TYPE      ptr;
{

#ifdef      DEBUG
printf("In traverse_arglist\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
```

```
                traverse_typeid(ptr->
                        RULE.BRANCH_arglist_1.typeid_1);
                break;
        case 2:     /*It is alternative  no 2    */
                traverse_arglist(ptr->
                        RULE.BRANCH_arglist_2.arglist_1);
                switch(ptr->RULE.BRANCH_arglist_2.OPR_2->type)
                {
                        case 1:     /*It is an operator    */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_typeid(ptr->
                        RULE.BRANCH_arglist_2.typeid_3);
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_arglist\n");
#endif

}/* end traverse_arglist*/



/*Routine to walk a mtuple branch of the syntax tree*/

traverse_mtuple(ptr)
MTUPLE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_mtuple\n");
#endif

    switch(ptr->type)
    {
        case 1:     /*It is alternative  no 1    */
                traverse_expr(ptr->RULE.BRANCH_mtuple_1.expr_1);
                break;
        case 2:     /*It is alternative  no 2    */
                traverse_mtuple(ptr->
                        RULE.BRANCH_mtuple_2.mtuple_1);
                switch(ptr->RULE.BRANCH_mtuple_2.OPR_2->type)
                {
                        case 1:     /*It is an operator    */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_expr(ptr->RULE.BRANCH_mtuple_2.expr_3);
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_mtuple\n");
#endif

}/* end traverse_mtuple*/
```

```
/*Routine to walk a stuple branch of the syntax tree*/

traverse_stuple(ptr)
STUPLE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_stuple\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_singleton(ptr->
                        RULE.BRANCH_stuple_1.singleton_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_stuple(ptr->
                        RULE.BRANCH_stuple_2.stuple_1);
                  switch(ptr->RULE.BRANCH_stuple_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_singleton(ptr->
                        RULE.BRANCH_stuple_2.singleton_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_stuple\n");
#endif

}/* end traverse_stuple*/



/*Routine to walk a singlist branch of the syntax tree*/

traverse_singlist(ptr)
SINGLIST_TYPE     ptr;
{

#ifdef      DEBUG
printf("In traverse_singlist\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_singlist_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
```

```
                traverse_singleton(ptr->
                        RULE.BRANCH_singlist_1.singleton_2);
                break;
        case 2:     /*It is alternative  no 2    */
                traverse_singlist(ptr->
                        RULE.BRANCH_singlist_2.singlist_1);
                switch(ptr->RULE.BRANCH_singlist_2.OPR_2->type)
                {
                        case 1:     /*It is an operator    */
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                traverse_singleton(ptr->
                        RULE.BRANCH_singlist_2.singleton_3);
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_singlist\n");
#endif

}/* end traverse_singlist*/



/*Routine to walk a expr branch of the syntax tree*/

traverse_expr(ptr)
EXPR_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_expr\n");
#endif

    switch(ptr->type)
    {
        case 1:     /*It is alternative  no 1    */
                traverse_set(ptr->RULE.BRANCH_expr_1.set_1);
                break;
        case 2:     /*It is alternative  no 2    */
                traverse_singleton(ptr->
                        RULE.BRANCH_expr_2.singleton_1);
                break;
        default:printf("ERROR: illegal alternative no.\n");
    }/* end switch */

#ifdef      DEBUG
printf("Out traverse_expr\n");
#endif

}/* end traverse_expr*/



/*Routine to walk a imperative branch of the syntax tree*/

traverse_imperative(ptr)
IMPERATIVE_TYPE   ptr;
{
```

```
#ifdef      DEBUG
printf("In traverse_imperative\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_forloop(ptr->
                        RULE.BRANCH_imperative_1.forloop_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_gpimperative(ptr->
                        RULE.BRANCH_imperative_2.gpimperative_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_imperative\n");
#endif

}/* end traverse_imperative*/



/*Routine to walk a forloop branch of the syntax tree*/

traverse_forloop(ptr)
FORLOOP_TYPE      ptr;
{

#ifdef      DEBUG
printf("In traverse_forloop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_forloop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_forloop_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_set(ptr->RULE.BRANCH_forloop_1.set_3);
                  traverse_imperative(ptr->
                        RULE.BRANCH_forloop_1.imperative_4);
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_forloop_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
```

```
                }/* end switch */
                traverse_singleton(ptr->
                      RULE.BRANCH_forloop_2.singleton_2);
                traverse_imperative(ptr->
                      RULE.BRANCH_forloop_2.imperative_3);
                break;
          default:printf("ERROR: illegal alternative no.\n");
     }/* end switch */

#ifdef      DEBUG
printf("Out traverse_forloop\n");
#endif

}/* end traverse_forloop*/



/*Routine to walk a gpimperative branch of the syntax tree*/

traverse_gpimperative(ptr)
GPIMPERATIVE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_gpimperative\n");
#endif

     switch(ptr->type)
     {
          case 1:     /*It is alternative  no 1     */
                switch(ptr->RULE.BRANCH_gpimperative_1.OPR_1->
                      type)
                {
                      case 1:     /*It is an operator     */
                            break;
                      default:printf("ERROR : Wrong lexeme type
                            generated\n");
                }/* end switch */
                traverse_stuple(ptr->
                      RULE.BRANCH_gpimperative_1.stuple_2);
                break;
          default:printf("ERROR: illegal alternative no.\n");
     }/* end switch */

#ifdef      DEBUG
printf("Out traverse_gpimperative\n");
#endif

}/* end traverse_gpimperative*/



/*Routine to walk a set branch of the syntax tree*/

traverse_set(ptr)
SET_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_set\n");
#endif

     switch(ptr->type)
```

```
{
        case 1:     /*It is alternative  no 1    */
              traverse_typeid(ptr->RULE.BRANCH_set_1.typeid_1);
              if (phase == 0)
              check_vblid(ptr->RULE.BRANCH_set_1.typeid_1);
              break;
        case 2:     /*It is alternative  no 2    */
              traverse_mvfuncall(ptr->
                    RULE.BRANCH_set_2.mvfuncall_1);
              break;
        case 3:     /*It is alternative  no 3    */
              switch(ptr->RULE.BRANCH_set_3.OPR_1->type)
              {
                    case 1:     /*It is an operator    */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                    generated\n");
              }/* end switch */
              traverse_stuple(ptr->RULE.BRANCH_set_3.stuple_2);
              switch(ptr->RULE.BRANCH_set_3.OPR_3->type)
              {
                    case 1:     /*It is an operator    */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                    generated\n");
              }/* end switch */
              break;
        case 4:     /*It is alternative  no 4    */
              switch(ptr->RULE.BRANCH_set_4.OPR_1->type)
              {
                    case 1:     /*It is an operator    */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                    generated\n");
              }/* end switch */
              traverse_set(ptr->RULE.BRANCH_set_4.set_2);
              switch(ptr->RULE.BRANCH_set_4.OPR_3->type)
              {
                    case 1:     /*It is an operator    */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                          generated\n");
              }/* end switch */
              break;
        case 5:     /*It is alternative  no 5    */
              switch(ptr->RULE.BRANCH_set_5.IDENTIFIER_1->type)
              {
                    case 2:     /*It is an identifier  */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                          generated\n");
              }/* end switch */
              switch(ptr->RULE.BRANCH_set_5.OPR_2->type)
              {
                    case 1:     /*It is an operator    */
                          break;
                    default:printf("ERROR : Wrong lexeme type
                    generated\n");
              }/* end switch */
              traverse_set(ptr->RULE.BRANCH_set_5.set_3);
              break;
        case 6:     /*It is alternative  no 6    */
              traverse_set(ptr->RULE.BRANCH_set_6.set_1);
```

```
                        switch(ptr->RULE.BRANCH_set_6.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                              generated\n");
                        }/* end switch */
                        traverse_typeid(ptr->RULE.BRANCH_set_6.typeid_3);
                        break;
                  case 7:     /*It is alternative  no 7     */
                        traverse_set(ptr->RULE.BRANCH_set_7.set_1);
                        traverse_comp(ptr->RULE.BRANCH_set_7.comp_2);
                        traverse_singleton(ptr->
                              RULE.BRANCH_set_7.singleton_3);
                        break;
                  case 8:     /*It is alternative  no 8     */
                        traverse_set(ptr->RULE.BRANCH_set_8.set_1);
                        traverse_comp(ptr->RULE.BRANCH_set_8.comp_2);
                        traverse_quant(ptr->RULE.BRANCH_set_8.quant_3);
                        traverse_set(ptr->RULE.BRANCH_set_8.set_4);
                        break;
                  case 9:     /*It is alternative  no 9     */
                        traverse_set(ptr->RULE.BRANCH_set_9.set_1);
                        switch(ptr->RULE.BRANCH_set_9.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        switch(ptr->RULE.BRANCH_set_9.OPR_3->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_pred(ptr->RULE.BRANCH_set_9.pred_4);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_set\n");
#endif

}/* end traverse_set*/



/*Routine to walk a pred branch of the syntax tree*/

traverse_pred(ptr)
PRED_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_pred\n");
#endif

      if (temppred == NULL)
            temppred = ptr;
      switch(ptr->type)
```

```
        {
                case 1:     /*It is alternative  no 1     */
                        switch(ptr->RULE.BRANCH_pred_1.OPR_1->type)
                        {
                                case 1:     /*It is an operator     */
                                        break;
                                default:printf("ERROR : Wrong lexeme type
                                        generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                                RULE.BRANCH_pred_1.singleton_2);
                        traverse_pred(ptr->RULE.BRANCH_pred_1.pred_3);
                        break;
                case 2:     /*It is alternative  no 2     */
                        traverse_singleton(ptr->
                                RULE.BRANCH_pred_2.singleton_1);
                        break;
                case 3:     /*It is alternative  no 3     */
                        switch(ptr->RULE.BRANCH_pred_3.OPR_1->type)
                        {
                                case 1:     /*It is an operator     */
                                        break;
                                default:printf("ERROR : Wrong lexeme type
                                        generated\n");
                        }/* end switch */
                        traverse_quant(ptr->RULE.BRANCH_pred_3.quant_2);
                        traverse_set(ptr->RULE.BRANCH_pred_3.set_3);
                        traverse_pred(ptr->RULE.BRANCH_pred_3.pred_4);
                        break;
                case 4:     /*It is alternative  no 4     */
                        traverse_singleton(ptr->
                                RULE.BRANCH_pred_4.singleton_1);
                        traverse_comp(ptr->RULE.BRANCH_pred_4.comp_2);
                        traverse_singleton(ptr->
                                RULE.BRANCH_pred_4.singleton_3);
                        break;
                case 5:     /*It is alternative  no 5     */
                        traverse_singleton(ptr->
                                RULE.BRANCH_pred_5.singleton_1);
                        traverse_comp(ptr->RULE.BRANCH_pred_5.comp_2);
                        traverse_quant(ptr->RULE.BRANCH_pred_5.quant_3);
                        traverse_set(ptr->RULE.BRANCH_pred_5.set_4);
                        break;
                case 6:     /*It is alternative  no 6     */
                        traverse_quant(ptr->RULE.BRANCH_pred_6.quant_1);
                        traverse_set(ptr->RULE.BRANCH_pred_6.set_2);
                        traverse_comp(ptr->RULE.BRANCH_pred_6.comp_3);
                        traverse_quant(ptr->RULE.BRANCH_pred_6.quant_4);
                        traverse_set(ptr->RULE.BRANCH_pred_6.set_5);
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_pred\n");
#endif

}/* end traverse_pred*/


/*Routine to walk a singleton branch of the syntax tree*/
```

```
traverse_singleton(ptr)
SINGLETON_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_singleton\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  traverse_expl(ptr->
                        RULE.BRANCH_singleton_1.expl_1);
                  break;
            case 2:     /*It is alternative  no 2    */
                  traverse_expl(ptr->
                        RULE.BRANCH_singleton_2.expl_1);
                  traverse_orexpl(ptr->
                        RULE.BRANCH_singleton_2.orexpl_2);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_singleton\n");
#endif

}/* end traverse_singleton*/



/*Routine to walk a orexpl branch of the syntax tree*/

traverse_orexpl(ptr)
OREXPl_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_orexpl\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  switch(ptr->RULE.BRANCH_orexpl_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator    */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                  }/* end switch */
                  traverse_expl(ptr->RULE.BRANCH_orexpl_1.expl_2);
                  break;
            case 2:     /*It is alternative  no 2    */
                  traverse_orexpl(ptr->
                        RULE.BRANCH_orexpl_2.orexpl_1);
                  switch(ptr->RULE.BRANCH_orexpl_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator    */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                  }/* end switch */
```

```
                        traverse_expl(ptr->RULE.BRANCH_orexpl_2.expl_3);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_orexpl\n");
#endif

}/* end traverse_orexpl*/



/*Routine to walk a expl branch of the syntax tree*/

traverse_expl(ptr)
EXPl_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_expl\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_exp2(ptr->RULE.BRANCH_expl_1.exp2_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_exp2(ptr->RULE.BRANCH_expl_2.exp2_1);
                  traverse_andexp2(ptr->
                        RULE.BRANCH_expl_2.andexp2_2);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_expl\n");
#endif

}/* end traverse_expl*/



/*Routine to walk a andexp2 branch of the syntax tree*/

traverse_andexp2(ptr)
ANDEXP2_TYPE      ptr;
{

#ifdef      DEBUG
printf("In traverse_andexp2\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_andexp2_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
```

```
                    }/* end switch */
                    traverse_exp2(ptr->RULE.BRANCH_andexp2_1.exp2_2);
                    break;
              case 2:     /*It is alternative  no 2     */
                    traverse_andexp2(ptr->
                          RULE.BRANCH_andexp2_2.andexp2_1);
                    switch(ptr->RULE.BRANCH_andexp2_2.OPR_2->type)
                    {
                          case 1:     /*It is an operator     */
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    traverse_exp2(ptr->RULE.BRANCH_andexp2_2.exp2_3);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_andexp2\n");
#endif

}/* end traverse_andexp2*/



/*Routine to walk a exp2 branch of the syntax tree*/

traverse_exp2(ptr)
EXP2_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp2\n");
#endif

        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    traverse_exp3(ptr->RULE.BRANCH_exp2_1.exp3_1);
                    break;
              case 2:     /*It is alternative  no 2     */
                    switch(ptr->RULE.BRANCH_exp2_2.OPR_1->type)
                    {
                          case 1:     /*It is an operator     */
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    traverse_exp3(ptr->RULE.BRANCH_exp2_2.exp3_2);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp2\n");
#endif

}/* end traverse_exp2*/



/*Routine to walk a exp3 branch of the syntax tree*/
```

```
traverse_exp3(ptr)
EXP3_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp3\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_exp4(ptr->RULE.BRANCH_exp3_1.exp4_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_exp4(ptr->RULE.BRANCH_exp3_2.exp4_1);
                  traverse_comp(ptr->RULE.BRANCH_exp3_2.comp_2);
                  traverse_exp4(ptr->RULE.BRANCH_exp3_2.exp4_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp3\n");
#endif

}/* end traverse_exp3*/



/*Routine to walk a exp4 branch of the syntax tree*/

traverse_exp4(ptr)
EXP4_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp4\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_exp5(ptr->RULE.BRANCH_exp4_1.exp5_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_exp5(ptr->RULE.BRANCH_exp4_2.exp5_1);
                  traverse_addop(ptr->RULE.BRANCH_exp4_2.addop_2);
                  traverse_exp5(ptr->RULE.BRANCH_exp4_2.exp5_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp4\n");
#endif

}/* end traverse_exp4*/



/*Routine to walk a exp5 branch of the syntax tree
*/
```

```
traverse_exp5(ptr)
EXP5_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp5\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_exp6(ptr->RULE.BRANCH_exp5_1.exp6_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_exp6(ptr->RULE.BRANCH_exp5_2.exp6_1);
                  traverse_mulop(ptr->RULE.BRANCH_exp5_2.mulop_2);
                  traverse_exp6(ptr->RULE.BRANCH_exp5_2.exp6_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp5\n");
#endif

}/* end traverse_exp5*/



/*Routine to walk a exp6 branch of the syntax tree*/

traverse_exp6(ptr)
EXP6_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp6\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  traverse_exp7(ptr->RULE.BRANCH_exp6_1.exp7_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  traverse_exp7(ptr->RULE.BRANCH_exp6_2.exp7_1);
                  switch(ptr->RULE.BRANCH_exp6_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_typeid(ptr->
                        RULE.BRANCH_exp6_2.typeid_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp6\n");
#endif
```

```
}/* end traverse_exp6*/


/*Routine to walk a exp7 branch of the syntax tree*/

traverse_exp7(ptr)
EXP7_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_exp7\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  traverse_constant(ptr->
                        RULE.BRANCH_exp7_1.constant_1);
                  break;
            case 2:     /*It is alternative  no 2    */
                  traverse_vblid(ptr->RULE.BRANCH_exp7_2.vblid_1);
                  if (phase == 0)
                  check_vblid(ptr->RULE.BRANCH_exp7_2.vblid_1);
                  break;
            case 3:     /*It is alternative  no 3    */
                  traverse_mvfuncall(ptr->
                        RULE.BRANCH_exp7_3.mvfuncall_1);
                  break;
            case 4:     /*It is alternative  no 4    */
                  traverse_aggcall(ptr->
                        RULE.BRANCH_exp7_4.aggcall_1);
                  break;
            case 5:     /*It is alternative  no 5    */
                  traverse_quant(ptr->RULE.BRANCH_exp7_5.quant_1);
                  traverse_set(ptr->RULE.BRANCH_exp7_5.set_2);
                  switch(ptr->RULE.BRANCH_exp7_5.OPR_3->type)
                  {
                        case 1:     /*It is an operator    */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_singleton(ptr->
                        RULE.BRANCH_exp7_5.singleton_4);
                  break;
            case 6:     /*It is alternative  no 6    */
                  switch(ptr->RULE.BRANCH_exp7_6.OPR_1->type)
                  {
                        case 1:     /*It is an operator    */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_set(ptr->RULE.BRANCH_exp7_6.set_2);
                  break;
            case 7:     /*It is alternative  no 7    */
                  switch(ptr->RULE.BRANCH_exp7_7.OPR_1->type)
                  {
                        case 1:     /*It is an operator    */
                              break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                              RULE.BRANCH_exp7_7.singleton_2);
                        switch(ptr->RULE.BRANCH_exp7_7.OPR_3->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_exp7\n");
#endif

}/* end traverse_exp7*/



/*Routine to walk a aggcall branch of the syntax tree*/

traverse_aggcall(ptr)
AGGCALL_TYPE      ptr;
{

#ifdef      DEBUG
printf("In traverse_aggcall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_set(ptr->RULE.BRANCH_aggcall_1.set_3);
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_aggcall_2.OPR_1->type)
                  {
```

```
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_aggcall_2.OPR_2->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        traverse_set(ptr->RULE.BRANCH_aggcall_2.set_3);
        switch(ptr->RULE.BRANCH_aggcall_2.OPR_4->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        break;
case 3:     /*It is alternative  no 3     */
        switch(ptr->RULE.BRANCH_aggcall_3.OPR_1->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_aggcall_3.OPR_2->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        traverse_set(ptr->RULE.BRANCH_aggcall_3.set_3);
        switch(ptr->RULE.BRANCH_aggcall_3.OPR_4->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        break;
case 4:     /*It is alternative  no 4     */
        switch(ptr->RULE.BRANCH_aggcall_4.OPR_1->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_aggcall_4.OPR_2->type)
        {
                case 1:     /*It is an operator     */
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
        }/* end switch */
        traverse_singleton(ptr->
                RULE.BRANCH_aggcall_4.singleton_3);
        switch(ptr->RULE.BRANCH_aggcall_4.OPR_4->type)
```

```
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        traverse_mtuple(ptr->
              RULE.BRANCH_aggcall_4.mtuple_5);
        switch(ptr->RULE.BRANCH_aggcall_4.OPR_6->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 5:     /*It is alternative  no 5     */
        switch(ptr->RULE.BRANCH_aggcall_5.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_aggcall_5.OPR_2->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        traverse_singleton(ptr->
              RULE.BRANCH_aggcall_5.singleton_3);
        switch(ptr->RULE.BRANCH_aggcall_5.OPR_4->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        traverse_mtuple(ptr->
              RULE.BRANCH_aggcall_5.mtuple_5);
        switch(ptr->RULE.BRANCH_aggcall_5.OPR_6->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 6:     /*It is alternative  no 6     */
        switch(ptr->RULE.BRANCH_aggcall_6.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_aggcall_6.OPR_2->type)
        {
              case 1:     /*It is an operator     */
                    break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                              RULE.BRANCH_aggcall_6.singleton_3);
                        switch(ptr->RULE.BRANCH_aggcall_6.OPR_4->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  case 7:     /*It is alternative  no 7     */
                        switch(ptr->RULE.BRANCH_aggcall_7.OPR_1->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        switch(ptr->RULE.BRANCH_aggcall_7.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                              RULE.BRANCH_aggcall_7.singleton_3);
                        switch(ptr->RULE.BRANCH_aggcall_7.OPR_4->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_mtuple(ptr->
                              RULE.BRANCH_aggcall_7.mtuple_5);
                        switch(ptr->RULE.BRANCH_aggcall_7.OPR_6->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_aggcall\n");
#endif

}/* end traverse_aggcall*/



/*Routine to walk a comp branch of the syntax tree*/

traverse_comp(ptr)
COMP_TYPE   ptr;
{
```

```
#ifdef      DEBUG
printf("In traverse_comp\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_comp_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_comp_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 3:     /*It is alternative  no 3     */
                  switch(ptr->RULE.BRANCH_comp_3.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 4:     /*It is alternative  no 4     */
                  switch(ptr->RULE.BRANCH_comp_4.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 5:     /*It is alternative  no 5     */
                  switch(ptr->RULE.BRANCH_comp_5.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 6:     /*It is alternative  no 6     */
                  switch(ptr->RULE.BRANCH_comp_6.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */
```

```
#ifdef      DEBUG
printf("Out traverse_comp\n");
#endif

}/* end traverse_comp*/


/*Routine to walk a quant branch of the syntax tree*/

traverse_quant(ptr)
QUANT_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_quant\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_quant_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_quant_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 3:     /*It is alternative  no 3     */
                  switch(ptr->RULE.BRANCH_quant_3.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 4:     /*It is alternative  no 4     */
                  switch(ptr->RULE.BRANCH_quant_4.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_leastmost(ptr->
                        RULE.BRANCH_quant_4.leastmost_2);
                  traverse_singleton(ptr->
                        RULE.BRANCH_quant_4.singleton_3);
                  break;
            case 5:     /*It is alternative  no 5     */
                  switch(ptr->RULE.BRANCH_quant_5.OPR_1->type)
```

```
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                                    RULE.BRANCH_quant_5.singleton_2);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_quant\n");
#endif

}/* end traverse_quant*/



/*Routine to walk a leastmost branch of the syntax tree*/

traverse_leastmost(ptr)
LEASTMOST_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_leastmost\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_leastmost_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_leastmost_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_leastmost\n");
#endif

}/* end traverse_leastmost*/



/*Routine to walk a addop branch of the syntax tree*/
```

```
traverse_addop(ptr)
ADDOP_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_addop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_addop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_addop_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_addop\n");
#endif

}/* end traverse_addop*/



/*Routine to walk a mulop branch of the syntax tree*/

traverse_mulop(ptr)
MULOP_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_mulop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_mulop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_mulop_2.OPR_1->type)
                  {
```

```
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_mulop\n");
#endif

}/* end traverse_mulop*/



/*Routine to walk a constant branch of the syntax tree*/

traverse_constant(ptr)
CONSTANT_TYPE     ptr;
{

#ifdef      DEBUG
printf("In traverse_constant\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_constant_1.INTEGER_1->
                        type)
                  {
                        case 3:     /*It is an integer      */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_constant_2.STRING_1->
                        type)
                  {
                        case 4:     /*It is a string        */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 3:     /*It is alternative  no 3     */
                  traverse_bool(ptr->
                        RULE.BRANCH_constant_3.bool_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_constant\n");
#endif

}/* end traverse_constant*/
```

```
/*Routine to walk a bool branch of the syntax tree*/

traverse_bool(ptr)
BOOL_TYPE   ptr;
{

#ifdef      DEBUG
printf("In traverse_bool\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_bool_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->RULE.BRANCH_bool_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_bool\n");
#endif

}/* end traverse_bool*/



/*Routine to walk a funcid branch of the syntax tree*/

traverse_funcid(ptr)
FUNCID_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_funcid\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
                        type)
                  {
                        case 6:     /*It is an identifier   */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
```

```
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_funcid\n");
#endif

}/* end traverse_funcid*/



/*Routine to walk a typeid branch of the syntax tree*/

traverse_typeid(ptr)
TYPEID_TYPE       ptr;
{

#ifdef      DEBUG
printf("In traverse_typeid\n");
#endif

        if (tempvblid == NULL)
        tempvblid = (VBLID_TYPE) ptr;
        switch(ptr->type)
        {
                case 1:     /*It is alternative  no 1     */
                        switch(ptr->RULE.BRANCH_typeid_1.IDENTIFIER_1->
                                type)
                        {
                                case 2:     /*It is an identifier   */
                                        break;
                                default:printf("ERROR : Wrong lexeme type
                                        generated\n");
                        }/* end switch */
                        break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out traverse_typeid\n");
#endif

}/* end traverse_typeid*/



/*Routine to walk a vblid branch of the syntax tree*/

traverse_vblid(ptr)
VBLID_TYPE  ptr;
{

#ifdef      DEBUG
printf("In traverse_vblid\n");
#endif

        switch(ptr->type)
        {
                case 1:     /*It is alternative  no 1     */
                        switch(ptr->RULE.BRANCH_vblid_1.IDENTIFIERV_1->
                                type)
                        {
```

```
                        case 5:     /*It is an var identifier     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out traverse_vblid\n");
#endif

}/* end traverse_vblid*/



/*Routine to walk a svfuncall branch of the syntax tree*/

traverse_svfuncall(ptr)
SVFUNCALL_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_svfuncall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->
                        RULE.BRANCH_svfuncall_1.IDENTIFIERF_1->
                              type)
                  {
                        case 6:     /*It is an identifier   */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  comp_funcid(ptr->
                        RULE.BRANCH_svfuncall_1.IDENTIFIERF_1->
                              textval.s);
                  switch(ptr->RULE.BRANCH_svfuncall_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  traverse_singleton(ptr->
                        RULE.BRANCH_svfuncall_1.singleton_3);
                  switch(ptr->RULE.BRANCH_svfuncall_1.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  switch(ptr->
                        RULE.BRANCH_svfuncall_2.IDENTIFIERF_1->
                              type)
```

```
                        {
                              case 6:     /*It is an identifier   */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        comp_funcid(ptr->
                              RULE.BRANCH_svfuncall_2.IDENTIFIERF_1->
                                    textval.s);
                        switch(ptr->RULE.BRANCH_svfuncall_2.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_singleton(ptr->
                              RULE.BRANCH_svfuncall_2.singleton_3);
                        traverse_singlist(ptr->
                              RULE.BRANCH_svfuncall_2.singlist_4);
                        switch(ptr->RULE.BRANCH_svfuncall_2.OPR_5->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_svfuncall\n");
#endif

}/* end traverse_svfuncall*/



/*Routine to walk a mvfuncall branch of the syntax tree*/

traverse_mvfuncall(ptr)
MVFUNCALL_TYPE    ptr;
{

#ifdef      DEBUG
printf("In traverse_mvfuncall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  switch(ptr->
                        RULE.BRANCH_mvfuncall_1.IDENTIFIERF_1->
                              type)
                  {
                        case 6:     /*It is an identifier   */
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
```

```
                        comp_funcid(ptr->
                              RULE.BRANCH_mvfuncall_1.IDENTIFIERF_1->
                                    textval.s);
                        switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        traverse_mtuple(ptr->
                              RULE.BRANCH_mvfuncall_1.mtuple_3);
                        switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_4->type)
                        {
                              case 1:     /*It is an operator     */
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out traverse_mvfuncall\n");
#endif

}/* end traverse_mvfuncall*/



/*Routine to write out the current totals to a file*/

write_totals()
{
      FILE *dp;

#ifdef      DEBUG
printf("In write_totals\n");
#endif

      if ((dp = fopen("FQLFE_counts", "w")) == NULL)
      {
            printf("FQLFE:Unable to open file FQLFE_counts\n");
            exit(1);
      }
      fprintf(dp, "%d %d %d", basetot, nbtot, dertot);
      fclose(dp);

#ifdef      DEBUG
printf("Out write_totals\n");
#endif

}/* end write_totals*/
```

```
/*          FILE: semantic.c  */


/* These routines do the semantic checking on the syntax tree of
the users input   */

/* includes for this file     */

#include          <stdio.h>
#include          <stdio.h>
#include          "tabledefs.h"
#include          "trans.h"

/* externals for this file    */

extern            FILE              *fp;
extern            int               relcount;
extern            int               basetot;
extern            int               nbtot;
extern            int               dertot;
extern            int               linenum;
extern            DER_DETAIL        *derptr;
extern            BASE_DETAIL       *bptr;
extern            NB_DETAIL         *nbptr;
extern            DER_DETAIL        der_detail;
extern            BASE_DETAIL       base_detail;
extern            NB_DETAIL         nb_detail;
extern            CATALOG           catalog;
extern            RANGELIST         rangelist;
extern            int               found;
extern            output_mtuple();

/* locals for this file       */
#define NULL 0
int incheckvblid = 0;



/*This function checks the funcid s against the relational
description the name s must be the name of a relation in the
relational database andtherefore should appear in the table
catalog.*/

check_funcid(s)
char *s;
{
        int i, true;

#ifdef      DEBUG
        printf("In check_funcid\n");
#endif

        true = 0;
        for (i = 0; i < relcount; i++)
        {
                if (strcmp(s, catalog.relation[i].relname) == 0)
                        true = 1;
        }
        if (!true)
                        execerror(s, "invalid function name");

#ifdef      DEBUG
        printf("Out check_funcid\n");
```

```
#endif

}


/*This function stores the name of a base function in the table
bfunctable. All base functions are of type ENTITY*/

store_bfuncid(s)
char *s;
{
      int i,true;
      BASE_DETAIL       *ptr;

#ifdef      DEBUG
      printf("In store_bfuncid\n");
#endif

      ptr = &base_detail;

      true = 0;
      while (ptr->nextfunc != NULL)
      {
            /*check if function is already in table */
            ptr = ptr->nextfunc;
                if ((strcmp(ptr->funcname, s)) == 0)
                        true = 1;
      }

      /* check value of true */

      if (true)       /* ie function already exists */
      {
            execerror(s, "already exists as base function");
            return;
      }

      /* otherwise current value of ptr indicates position in
      table where s should be entered */

      ptr->nextfunc = (BASE_DETAIL *) malloc(sizeof(BASE_DETAIL));
      ptr = ptr->nextfunc;
      strcpy(ptr->funcname, s);
      ptr->nextfunc = NULL;

/* copy s into table */

/*increment the count of total base functions */

      basetot++;

#ifdef      DEBUG
        printf("Out store_bfuncid\n");
#endif


}


/*This function checks the typeid s identifier is a valid one. i.e
it is the name of an identifier in the base entity table.*/
```

```
check_typeid(s)
char *s;
{
      int i, true;
      BASE_DETAIL       *ptr;
      RANGELIST   *rptr;

#ifdef      DEBUG
        printf("In check_typeid\n");
#endif

      true = 0;
      ptr = &base_detail;
      while (ptr->nextfunc != NULL)
        {
            ptr = ptr->nextfunc;
                if (strcmp(ptr->funcname, s) == 0)
                {
                        true = 1;
                }
        }
      /* the rangelist needs to be checked since typeid may be a
            range variable this needs to be done only if
            this function was called by check_vblid */
      if (incheckvblid)
      {
            rptr = &rangelist;
            while (rptr->next != NULL)
            {
                  rptr = rptr->next;
                  if (strcmp(rptr->range, s) == 0)
                  {
                              true = 1;
                  }
            }
            if (!true)
            {
                  if (strlen(s) <= 2)
                  /* assume a range variable */
                        true = 1;
            }
      }

      if (!true)
      {
            execerror(s, "typeid is not a base function");
            return;
      }
      /* otherwise assume that typeid does exist as an entity
                  in the table and therefore continue     */

#ifdef      DEBUG
        printf("Out check_typeid\n");
#endif

}



/*This function stores the name of a nonbase function. These
functions return either INT, STR. i.e. other base functions when
they are evaluated. The table is nbfunc_table.*/
```

```
store_nbfuncid(s,t,u)
char *s, *t, *u;
{
      int i, true;
      NB_DETAIL   *ptr;

#ifdef      DEBUG
        printf("In store_nbfuncid\n");
#endif

      ptr = &nb_detail;
      true = 0;
      while (ptr->nextfunc != NULL)
      {
                /*check if function is already in table */
            ptr = ptr->nextfunc;
                if ((strcmp(ptr->funcname, s)) == 0)
            {
                  if (strcmp(ptr->argname, t) == 0)
                  {
                        execerror(s, "already exists as non-base
                              function");
                        return;
                  }
            }
      }

      /* otherwise current value of ptr indicates position in
            table where s should be entered */

      ptr->nextfunc = (NB_DETAIL *) malloc(sizeof(NB_DETAIL));
      ptr = ptr->nextfunc;
      strcpy(ptr->funcname,s);
      strcpy(ptr->argname,t);
      ptr->nextfunc = NULL;

      /* copy s into table */

      /* Note the total count of non base functions
            needs to be incremented at this point in time.       */

      nbtot++;

#ifdef      DEBUG
        printf("Out store_nbfuncid\n");
#endif

}


/*This checks the funcid of a nonbase function against the
function table*/

check_nbfuncid(s,t,u)
char *s, *t, *u;
{

      int i, j, true;
#ifdef      DEBUG
        printf("In check_nbfuncid\n");
#endif
```

```
        /*check that funcid specifies an attribute of the
         relation specified by the argument also check that
         the return type is consistent with that attribute and
         also that the arg is a valid base entity. */

        /*s = funcname t = argument(ie relation u = return type */
        i = 0;
        true = 0;
        for (i = 0; i < relcount; i++)
        {
          true = 0;
          if (strcmp(catalog.relation[i].relname, t) == 0)
          {
            /* ie the arg t is a valid relation in the db */
            for (j = 0; j < catalog.relation[i].attrcount; j++)
            {
              if (strcmp(catalog.relation[i].attribute[j].attrname,
                    s) == 0)
              {
                /*ie the funcid s is a valid attribute of rel t */
              true = 1;
              if (strcmp(catalog.relation[i].attribute[j].attrtype,
                    u) != 0)
              {
                /* ie return type inconsistent */
                execerror(u, "inconsistent return type");
                    return;
              }
              else
              {
                    store_nbfuncid(s, t, u);
                    break;
              }
              }
            }
            if (!true)
              /* ie no attribute of name s in rel of name t */
              execerror(s, "invalid function (no attribute)");
            true = 1;
            break;
          }
        }
        if (!true)
            /* ie no rel of name t in relational db */
            execerror(t, "invalid argument (no relation)");

#ifdef      DEBUG
        printf("Out check_nbfuncid\n");
#endif

}



/*this does the checking for a function declared to return an
entity*/

check_entity_decl(ptr)
SIMPLE_DECL_TYPE ptr;
{

      FUNCID_TYPE s;
#ifdef      DEBUG
```

```
        printf("In check_entity\n");
#endif

      s = (ptr->RULE.BRANCH_simple_decl_1.funcid_2);
      check_funcid(s->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
            textval.s);
      store_bfuncid(s->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
            textval.s);

#ifdef      DEBUG
        printf("Out check_entity\n");
#endif

}



/*this does the checking for a function declared to return an
string type*/

check_str_decl(ptr)
SIMPLE_DECL_TYPE ptr;
{
      FUNCID_TYPE s;
      VBLID_TYPE t;

#ifdef      DEBUG
        printf("In check_str_decl\n");
#endif

      s = (FUNCID_TYPE)(ptr->RULE.BRANCH_simple_decl_2.funcid_2);
      t = (VBLID_TYPE)(ptr->RULE.BRANCH_simple_decl_2.vblid_4);
      check_nbfuncid(s->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
            textval.s,
                  t->RULE.BRANCH_vblid_1.IDENTIFIERV_1->textval.s,
                  ptr->RULE.BRANCH_simple_decl_2.OPR_7->textval.s);

#ifdef      DEBUG
        printf("Out check_str_decl\n");
#endif

}



/*this does the checking for a function declared to return an
integer type*/

check_int_decl(ptr)
SIMPLE_DECL_TYPE ptr;
{
      FUNCID_TYPE s;
      VBLID_TYPE t;

#ifdef      DEBUG
        printf("In check_int_decl\n");
#endif

      s = (FUNCID_TYPE)(ptr->RULE.BRANCH_simple_decl_3.funcid_2);
      t = (VBLID_TYPE)(ptr->RULE.BRANCH_simple_decl_3.vblid_4);
      check_nbfuncid(s->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
                  textval.s,
                  t->RULE.BRANCH_vblid_1.IDENTIFIERV_1->textval.s,
```

```
                ptr->RULE.BRANCH_simple_decl_3.OPR_7->textval.s);

#ifdef      DEBUG
        printf("Out check_str_decl\n");
#endif

}


/*this produces error messages during semantic analysis*/

execerror(s, t)
char *s, *t;
{
      printf("IN EXECERROR\n");
      printf("Statement number - %d: ", linenum - 1);
      printf("%s - %s\n", s, t);
}



check_inverse()
{
      printf("FQLFE: Sorry inverse functions currently
            unavailable\n");
}


check_trans()
{
      printf("FQLFE: Sorry transitive functions currently
            unavailable\n");
}



check_comp()
{
      printf("FQLFE: Sorry functions compound currently
            unavailable\n");
}



check_inter()
{
      printf("FQLFE: Sorry function intersection currently
            unavailable\n");
}

check_union()
{
      printf("FQLFE: Sorry function union currently
            unavailable\n");
}



check_diff()
{
      printf("FQLFE: Sorry function difference currently
            unavailable\n");
```

```
}


check_funcspec(s, t)
char *s;
MTUPLE_TYPE t;
{
      /* check derived function table to see if this function
            already exists. */

      int true;

#ifdef      DEBUG
        printf("In check_funcspec\n");
#endif

      true = 0;
      derptr = &der_detail;
      while (derptr->nextfunc != NULL)
        {
              /*check if function is already in table */
            derptr = derptr->nextfunc;
              if ((strcmp(derptr->funcname, s)) == 0)
            {
                        true = 1;
                break;
            }
        }

        /* check value of true */
      if (true) /*ie function already exists */
            /*walk mtuple for stored function and for
                  new function and compare them */
            if (compare_mtuple(derptr->argname, t)) /*1 if equal*/
            {
                  execerror(s, "This function already exists\n");
                  return 1;
            }

#ifdef      DEBUG
        printf("Out check_funcspec\n");
#endif

      return 0;

}/*end of check_funcspec */

check_comp_decl(ptr)
COMPLEX_DECL_TYPE      ptr;
{
      char *s;
      int *t, *v;
      /* s = ptr to funcid, t = ptr to mtuple, v= ptr to
definetypes */

#ifdef      DEBUG
        printf("In check_comp_decl\n");
#endif

      (int *)s =  (int *) (ptr->R
            RULE.BRANCH_complex_decl_1.funcspec_2);
```

```
        (int *)s =  (int *) (((FUNCSPEC_TYPE)s)->
              RULE.BRANCH_funcspec_1.funcid_1);
        s =  (((FUNCID_TYPE)s)->RULE.BRANCH_funcid_1.IDENTIFIERF_1
                    ->textval.s);

        t =  (int *) (ptr->RULE.BRANCH_complex_decl_1.funcspec_2);
        t =  (int *) (((FUNCSPEC_TYPE)t)->
              RULE.BRANCH_funcspec_1.mtuple_3);

        if (check_funcspec(s, t))
              return;

        v =  (int *) (ptr->
              RULE.BRANCH_complex_decl_1.definetypes_4);

        /* find end of table */

        /*skip to end of list to insert new func at end */

        derptr = &der_detail;
              while (derptr->nextfunc != NULL)
              derptr = derptr->nextfunc;

          /* otherwise current value of ptr indicates position in
                    table where function should be entered */

        derptr->nextfunc = (DER_DETAIL *)
              malloc(sizeof(DER_DETAIL));
        derptr = derptr->nextfunc;
        strcpy(derptr->funcname,s);
        derptr->argname = (MTUPLE_TYPE) t;
        derptr->result_type = (DEFINETYPES_TYPE) v;
        derptr->nextfunc = NULL;

        dertot++;

#ifdef      DEBUG
          printf("Out check_comp_decl\n");
#endif

}/* end of check_comp_decl */



compare_mtuple(ptr, nptr)
MTUPLE_TYPE ptr;
MTUPLE_TYPE nptr;
{

        int max = 10000;
        char arg1[10000], arg2[10000];
        /*ptr points to func in table, nptr points to new func */

#ifdef      DEBUG
          printf("In compare_mtuple\n");
#endif

        if ((fp = fopen("FQLFE_t1", "w")) == NULL)
        {
              printf("FQLFE:Unable to open file FQLFE_t1\n");
              exit(1);
        }
        output_mtuple(ptr);
```

```
        fclose(fp);
        if ((fp = fopen("FQLFE_t1", "r")) == NULL)
        {
              printf("FQLFE:Unable to open file FQLFE_t1\n");
              exit(1);
        }
        fgets(arg1, max, fp);
        fclose(fp);

        if ((fp = fopen("FQLFE_t2", "w")) == NULL)
        {
              printf("FQLFE:Unable to open file FQLFE_t2\n");
              exit(1);
        }
        output_mtuple(nptr);
        fclose(fp);
        if ((fp = fopen("FQLFE_t2", "r")) == NULL)
        {
              printf("FQLFE:Unable to open file FQLFE_t2\n");
              exit(1);
        }
        fgets(arg2, max, fp);
        fclose(fp);

#ifdef      DEBUG
          printf("Out compare_mtuple\n");
#endif

        /*now compare the two strings arg1 and arg2 */
        if ((strcmp(arg1, arg2) == 0) )
              return 1;
        else
              return 0;

}/*end of compare_mtuple*/

comp_funcid(s)
char *s;
{
        int i, true;
        NB_DETAIL   *ptr;
        DER_DETAIL  *derptr;

#ifdef      DEBUG
          printf("In comp_funcid\n");
#endif

        true = 0;
        ptr = &nb_detail;
        while (ptr->nextfunc != NULL)
          {
              ptr = ptr->nextfunc;
                   if (strcmp(ptr->funcname, s) == 0)
                   {
                           true = 1;
                     break;
                   }
          }
          if (!true)
          {
              derptr = &der_detail;
              while (derptr->nextfunc != NULL)
              {
```

```
                derptr = derptr->nextfunc;
                if (strcmp(derptr->funcname, s) == 0)
                {
                        true = 1;
                    break;
                }
            }
        }
        if (!true)
        {
                execerror(s, "is not a nonbase  or derived
                        function");
            return;
        }
        /* otherwise assume that funcid does exist as an entity
                in the table and therefore continue     */

#ifdef      DEBUG
        printf("Out comp_funcid\n");
#endif
}/* end of comp_funcid */



check_vblid(s)
VBLID_TYPE s;
{

#ifdef      DEBUG
        printf("In check_vblid\n");
#endif

      incheckvblid = 1;
      check_typeid(s->RULE.BRANCH_vblid_1.IDENTIFIERV_1->
            textval.s);
      incheckvblid = 0;

#ifdef      DEBUG
        printf("Out check_vblid\n");
#endif

}/*end of check_vblid */



der_lookup(ptr, mptr)
char *ptr;
MTUPLE_TYPE mptr;
{
      int i;

#ifdef      DEBUG
printf("In der_lookup\n");
#endif

      found = 0;
      derptr = &der_detail;
      for (i = 0; i < dertot; i++)
      {
            derptr = derptr->nextfunc;
            if (strcmp(derptr->funcname, ptr) == 0)
            {
                  if ((compare_mtuple(mptr, derptr->argname))
```

```
                                == 1)
                        {
                                found = 1;
                                break;
                        }
                }
        }

#ifdef      DEBUG
printf("Out der_lookup\n");
#endif

}
```

```
/*          FILE: tables.c    */


/* These routines to save the internal tables to file      */

/* includes for this file     */

#include          <stdio.h>
#include          "tabledefs.h"

/* externals for this file    */

extern            int               basetot, nbtot, dertot;
extern            DER_DETAIL        *derptr;
extern            BASE_DETAIL       *bptr;
extern            NB_DETAIL         *nbptr;
extern            DER_DETAIL        der_detail;
extern            BASE_DETAIL       base_detail;
extern            NB_DETAIL         nb_detail;
extern            FILE              *fp;



save_table()
{
      char *filename, *h;
      int i;
      char *space;
      char *getenv();

#ifdef      DEBUG
printf("In save_table\n");
#endif

      space = " ";
      if (( h = getenv("HOME")) == (char *) NULL)
      {
            printf("FQLFE: No home directory specified\n");
            exit(0);
      }

      /* sprintf(filename, "%s/%s", h, dbname_func); */
      if ((fp = fopen("dbname_func", "w")) == NULL)
            printf("FQLFE:Unable to access functional
                  description\n");

      fprintf(fp, "%d%s", basetot, space);
      fprintf(fp, "%d%s", nbtot, space);
      fprintf(fp, "%d%s", dertot, space);

      /* write out base table */
      bptr = &base_detail;
      for (i = 0; i < basetot; i++)
      {
            bptr = bptr->nextfunc;
            fputs(bptr->funcname, fp);
            fputs(space, fp);
      }

      /* write out nonbase table */
      nbptr = &nb_detail;
      for (i = 0; i < nbtot; i++)
```

```
        {
                nbptr = nbptr->nextfunc;
                fputs(nbptr->funcname, fp);
                fputs(space, fp);
                fputs(nbptr->argname, fp);
                fputs(space, fp);
                fputs(nbptr->result_type, fp);
                fputs(space, fp);
        }

        /* write out derived function table */
        derptr = &der_detail;
        for (i = 0; i < dertot; i++)
        {
                derptr = derptr->nextfunc;
                fputs(derptr->funcname, fp);
                fputs(space, fp);
                output_mtuple(derptr->argname);
                fputs(space, fp);
                output_definetypes(derptr->result_type);
                fputs(space, fp);
        }
        fclose(fp);

#ifdef      DEBUG
printf("Out save_table\n");
#endif
}/*end of save_table */
```

```
/*          FILE: output.c    */

/* Routines to traverse the syntax tree of the input & output it
to file     */

/* includes for this file     */

#include          <stdio.h>
#include          "structures.h"

/* externals for this file    */

extern            FILE        *fp;

/* locals for this file       */



/*routine to traverse a program branch and output it out to file*/

output_program(tree)

PROGRAM_TYPE      tree;
{
      PROGRAM_TYPE      ptr;
      int               *temp2;

#ifdef      DEBUG
printf("In output_program\n");
#endif

      ptr = tree;
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_statements(ptr->
                        RULE.BRANCH_program_1.statements_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */
      fclose(fp);

#ifdef      DEBUG
printf("Out output_program\n");
#endif

}/* end output_program*/



/*routine to traverse a statements branch and output it out to
file*/

output_statements(ptr)
STATEMENTS_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_statements\n");
#endif
```

```
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    output_statement(ptr->
                          RULE.BRANCH_statements_1.statement_1);
                    break;
              case 2:     /*It is alternative  no 2     */
                    fputs("2 ", fp);
                    output_statements(ptr->
                          RULE.BRANCH_statements_2.statements_1);
                    output_statement(ptr->
                          RULE.BRANCH_statements_2.statement_2);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_statements\n");
#endif

}/* end output_statements*/



/*routine to traverse a statement branch and output it out to
file*/

output_statement(ptr)
STATEMENT_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_statement\n");
#endif

        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    output_declarative(ptr->
                          RULE.BRANCH_statement_1.declarative_1);
                    break;
              case 2:     /*It is alternative  no 2     */
                    fputs("2 ", fp);
                    output_imperative(ptr->
                          RULE.BRANCH_statement_2.imperative_1);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_statement\n");
#endif

}/* end output_statement*/



/*routine to traverse a declarative branch and output it out to
file*/
```

```
output_declarative(ptr)
DECLARATIVE_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_declarative\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_simple_decl(ptr->
                        RULE.BRANCH_declarative_1.simple_decl_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_complex_decl(ptr->
                        RULE.BRANCH_declarative_2.complex_decl_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_declarative\n");
#endif

}/* end output_declarative*/



/*routine to traverse a simple_decl branch and output it out to
file*/

output_simple_decl(ptr)
SIMPLE_DECL_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_simple_decl\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_simple_decl_1.OPR_1->
                        type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_simple_decl_1.OPR_1->
                                    textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_funcid(ptr->
                        RULE.BRANCH_simple_decl_1.funcid_2);
                  switch(ptr->RULE.BRANCH_simple_decl_1.OPR_3->
                        type)
```

```
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_simple_decl_1.OPR_3->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_simple_decl_1.OPR_4->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_simple_decl_1.OPR_4->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_arrow(ptr->
              RULE.BRANCH_simple_decl_1.arrow_5);
        switch(ptr->RULE.BRANCH_simple_decl_1.OPR_6->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_simple_decl_1.OPR_6->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 2:     /*It is alternative  no 2     */
        fputs("2 ", fp);
        switch(ptr->RULE.BRANCH_simple_decl_2.OPR_1->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_simple_decl_2.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_funcid(ptr->
              RULE.BRANCH_simple_decl_2.funcid_2);
        switch(ptr->RULE.BRANCH_simple_decl_2.OPR_3->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
```

```
                        fputs(ptr->
                        RULE.BRANCH_simple_decl_1.OPR_3->
                              textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_vblid(ptr->
                  RULE.BRANCH_simple_decl_2.vblid_4);
            switch(ptr->RULE.BRANCH_simple_decl_2.OPR_5->
                  type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                        RULE.BRANCH_simple_decl_2.OPR_5->
                              textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_arrow(ptr->
                  RULE.BRANCH_simple_decl_2.arrow_6);
            switch(ptr->RULE.BRANCH_simple_decl_2.OPR_7->
                  type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                        RULE.BRANCH_simple_decl_2.OPR_7->
                              textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 3:     /*It is alternative  no 3     */
            fputs("3 ", fp);
            switch(ptr->RULE.BRANCH_simple_decl_3.OPR_1->
                  type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                        RULE.BRANCH_simple_decl_3.OPR_1->
                              textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_funcid(ptr->
                  RULE.BRANCH_simple_decl_3.funcid_2);
            switch(ptr->RULE.BRANCH_simple_decl_3.OPR_3->
                  type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
```

```
                                    fputs(ptr->
                                    RULE.BRANCH_simple_decl_3.OPR_3->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_vblid(ptr->
                              RULE.BRANCH_simple_decl_3.vblid_4);
                        switch(ptr->RULE.BRANCH_simple_decl_3.OPR_5->
                              type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_simple_decl_3.OPR_5->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_arrow(ptr->
                              RULE.BRANCH_simple_decl_3.arrow_6);
                        switch(ptr->RULE.BRANCH_simple_decl_3.OPR_7->
                              type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_simple_decl_3.OPR_7->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_simple_decl\n");
#endif

}/* end output_simple_decl*/



/*routine to traverse a complex_decl branch and output it out to
file
*/

output_complex_decl(ptr)
COMPLEX_DECL_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_complex_decl\n");
#endif

      switch(ptr->type)
```

```
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    switch(ptr->RULE.BRANCH_complex_decl_1.OPR_1->
                          type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                RULE.BRANCH_complex_decl_1.OPR_1->
                                      textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    output_funcspec(ptr->
                          RULE.BRANCH_complex_decl_1.funcspec_2);
                    output_arrow(ptr->
                          RULE.BRANCH_complex_decl_1.arrow_3);
                    output_definetypes(ptr->
                          RULE.BRANCH_complex_decl_1.definetypes_4);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_complex_decl\n");
#endif

}/* end output_complex_decl*/



/*routine to traverse a definetypes branch and output it out to
file*/

output_definetypes(ptr)
DEFINETYPES_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_definetypes\n");
#endif

        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    switch(ptr->RULE.BRANCH_definetypes_1.OPR_1->
                          type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                RULE.BRANCH_definetypes_1.OPR_1->
                                      textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
```

```
        switch(ptr->RULE.BRANCH_definetypes_1.OPR_2->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_1.OPR_2->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_funcspec(ptr->
              RULE.BRANCH_definetypes_1.funcspec_3);
        break;
case 2:     /*It is alternative  no 2     */
        fputs("2 ", fp);
        switch(ptr->RULE.BRANCH_definetypes_2.OPR_1->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_2.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_definetypes_2.OPR_2->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_2.OPR_2->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_expr(ptr->
              RULE.BRANCH_definetypes_2.expr_3);
        break;
case 3:     /*It is alternative  no 3     */
        fputs("3 ", fp);
        switch(ptr->RULE.BRANCH_definetypes_3.OPR_1->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_3.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
```

```
        switch(ptr->RULE.BRANCH_definetypes_3.OPR_2->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_3.OPR_2->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_mtuple(ptr->
              RULE.BRANCH_definetypes_3.mtuple_3);
        break;
case 4:     /*It is alternative  no 4     */
        fputs("4 ", fp);
        switch(ptr->RULE.BRANCH_definetypes_4.OPR_1->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_4.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        switch(ptr->RULE.BRANCH_definetypes_4.OPR_2->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_4.OPR_2->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        output_mtuple(ptr->
              RULE.BRANCH_definetypes_4.mtuple_3);
        break;
case 5:     /*It is alternative  no 5     */
        fputs("5 ", fp);
        switch(ptr->RULE.BRANCH_definetypes_5.OPR_1->
              type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                    RULE.BRANCH_definetypes_5.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
```

```
                        switch(ptr->RULE.BRANCH_definetypes_5.OPR_2->
                              type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_definetypes_5.OPR_2->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_mtuple(ptr->
                              RULE.BRANCH_definetypes_5.mtuple_3);
                        break;
                  case 6:     /*It is alternative  no 6     */
                        fputs("6 ", fp);
                        switch(ptr->RULE.BRANCH_definetypes_6.OPR_1->
                              type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->R
                                    RULE.BRANCH_definetypes_6.OPR_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        switch(ptr->RULE.BRANCH_definetypes_6.OPR_2->
                              type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_definetypes_6.OPR_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_mtuple(ptr->
                              RULE.BRANCH_definetypes_6.mtuple_3);
                        break;
                  case 7:     /*It is alternative  no 7     */
                        fputs("7 ", fp);
                        output_expr(ptr->
                              RULE.BRANCH_definetypes_7.expr_1);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_definetypes\n");
#endif

}/* end output_definetypes*/
```

```
/*routine to traverse a arrow branch and output it out to file*/

output_arrow(ptr)
ARROW_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_arrow\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_arrow_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_arrow_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  switch(ptr->RULE.BRANCH_arrow_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_arrow_2.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_arrow\n");
#endif

}/* end output_arrow*/



/*routine to traverse a funcspec branch and output it out to
file*/

output_funcspec(ptr)
FUNCSPEC_TYPE     ptr;
{

#ifdef      DEBUG
printf("In output_funcspec\n");
#endif
```

```
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    output_funcid(ptr->
                          RULE.BRANCH_funcspec_1.funcid_1);
                    switch(ptr->RULE.BRANCH_funcspec_1.OPR_2->type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                RULE.BRANCH_funcspec_1.OPR_2->
                                      textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    output_mtuple(ptr->
                          RULE.BRANCH_funcspec_1.mtuple_3);
                    switch(ptr->RULE.BRANCH_funcspec_1.OPR_4->type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                      RULE.BRANCH_funcspec_1.OPR_4->
                                            textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_funcspec\n");
#endif

}/* end output_funcspec*/


/*routine to traverse a arglist branch and output it out to file*/

output_arglist(ptr)
ARGLIST_TYPE      ptr;
{

#ifdef      DEBUG
printf("In output_arglist\n");
#endif

        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    output_typeid(ptr->
                          RULE.BRANCH_arglist_1.typeid_1);
                    break;
              case 2:     /*It is alternative  no 2     */
```

```
                        fputs("2 ", fp);
                        output_arglist(ptr->
                              RULE.BRANCH_arglist_2.arglist_1);
                        switch(ptr->RULE.BRANCH_arglist_2.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_arglist_2.OPR_2->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_typeid(ptr->
                              RULE.BRANCH_arglist_2.typeid_3);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_arglist\n");
#endif

}/* end output_arglist*/



/*routine to traverse a mtuple branch and output it out to file*/

output_mtuple(ptr)
MTUPLE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_mtuple\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_expr(ptr->RULE.BRANCH_mtuple_1.expr_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_mtuple(ptr->
                        RULE.BRANCH_mtuple_2.mtuple_1);
                  switch(ptr->RULE.BRANCH_mtuple_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_mtuple_2.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_expr(ptr->RULE.BRANCH_mtuple_2.expr_3);
                  break;
```

```
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_mtuple\n");
#endif

}/* end output_mtuple*/



/*routine to traverse a stuple branch and output it out to file*/

output_stuple(ptr)
STUPLE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_stuple\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_singleton(ptr->
                        RULE.BRANCH_stuple_1.singleton_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_stuple(ptr->
                        RULE.BRANCH_stuple_2.stuple_1);
                  switch(ptr->RULE.BRANCH_stuple_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_stuple_2.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_stuple_2.singleton_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_stuple\n");
#endif

}/* end output_stuple*/



/*routine to traverse a singlist branch and output it out to
file*/

output_singlist(ptr)
SINGLIST_TYPE     ptr;
```

```
{

#ifdef      DEBUG
printf("In output_singlist\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_singlist_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_singlist_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_singlist_1.singleton_2);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_singlist(ptr->
                        RULE.BRANCH_singlist_2.singlist_1);
                  switch(ptr->RULE.BRANCH_singlist_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_singlist_2.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_singlist_2.singleton_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
       }/* end switch */

#ifdef      DEBUG
printf("Out output_singlist\n");
#endif

}/* end output_singlist*/



/*routine to traverse a expr branch and output it out to file
*/

output_expr(ptr)
EXPR_TYPE   ptr;
{

#ifdef      DEBUG
```

```
printf("In output_expr\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_set(ptr->RULE.BRANCH_expr_1.set_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_singleton(ptr->
                        RULE.BRANCH_expr_2.singleton_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_expr\n");
#endif

}/* end output_expr*/



/*routine to traverse a imperative branch and output it out to
file*/

output_imperative(ptr)
IMPERATIVE_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_imperative\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_forloop(ptr->
                  RULE.BRANCH_imperative_1.forloop_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_gpimperative(ptr->R
                  RULE.BRANCH_imperative_2.gpimperative_1);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_imperative\n");
#endif

}/* end output_imperative*/



/*routine to traverse a forloop branch and output it out to file*/

output_forloop(ptr)
FORLOOP_TYPE      ptr;
```

```
{

#ifdef      DEBUG
printf("In output_forloop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_forloop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_forloop_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_forloop_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_forloop_1.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_set(ptr->RULE.BRANCH_forloop_1.set_3);
                  output_imperative(ptr->
                        RULE.BRANCH_forloop_1.imperative_4);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  switch(ptr->RULE.BRANCH_forloop_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_forloop_2.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_forloop_2.singleton_2);
                  output_imperative(ptr->
                        RULE.BRANCH_forloop_2.imperative_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_forloop\n");
#endif
```

```
}/* end output_forloop*/


/*routine to traverse a gpimperative branch and output it out to
file*/

output_gpimperative(ptr)
GPIMPERATIVE_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_gpimperative\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_gpimperative_1.OPR_1->
                        type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_gpimperative_1.OPR_1
                                    ->textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_stuple(ptr->
                        RULE.BRANCH_gpimperative_1.stuple_2);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_gpimperative\n");
#endif

}/* end output_gpimperative*/


/*routine to traverse a set branch and output it out to file*/

output_set(ptr)
SET_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_set\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_typeid(ptr->RULE.BRANCH_set_1.typeid_1);
                  break;
```

```
case 2:     /*It is alternative  no 2    */
      fputs("2 ", fp);
      output_mvfuncall(ptr->
            RULE.BRANCH_set_2.mvfuncall_1);
      break;
case 3:     /*It is alternative  no 3    */
      fputs("3 ", fp);
      switch(ptr->RULE.BRANCH_set_3.OPR_1->type)
      {
            case 1:     /*It is an operator    */
                  fputs("1 ", fp);
                  fputs(ptr->RULE.BRANCH_set_3.OPR_1->
                        textval.s, fp);
                  fputs(" ", fp);
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      output_stuple(ptr->RULE.BRANCH_set_3.stuple_2);
      switch(ptr->RULE.BRANCH_set_3.OPR_3->type)
      {
            case 1:     /*It is an operator    */
                  fputs("1 ", fp);
                  fputs(ptr->RULE.BRANCH_set_3.OPR_3->
                        textval.s, fp);
                  fputs(" ", fp);
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      break;
case 4:     /*It is alternative  no 4    */
      fputs("4 ", fp);
      switch(ptr->RULE.BRANCH_set_4.OPR_1->type)
      {
            case 1:     /*It is an operator    */
                  fputs("1 ", fp);
                  fputs(ptr->RULE.BRANCH_set_4.OPR_1->
                        textval.s, fp);
                  fputs(" ", fp);
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      output_set(ptr->RULE.BRANCH_set_4.set_2);
      switch(ptr->RULE.BRANCH_set_4.OPR_3->type)
      {
            case 1:     /*It is an operator    */
                  fputs("1 ", fp);
                  fputs(ptr->RULE.BRANCH_set_4.OPR_3->
                        textval.s, fp);
                  fputs(" ", fp);
                  break;
            default:printf("ERROR : Wrong lexeme type
                  generated\n");
      }/* end switch */
      break;
case 5:     /*It is alternative  no 5    */
      fputs("5 ", fp);
      switch(ptr->RULE.BRANCH_set_5.IDENTIFIER_1->type)
      {
            case 2:     /*It is an identifier  */
                  fputs("2 ", fp);
```

```
                fputs(ptr->
                      RULE.BRANCH_set_5.IDENTIFIER_1->
                            textval.s, fp);
                      fputs(" ", fp);
                      break;
                default:printf("ERROR : Wrong lexeme type
                      generated\n");
          }/* end switch */
          switch(ptr->RULE.BRANCH_set_5.OPR_2->type)
          {
                case 1:     /*It is an operator     */
                      fputs("1 ", fp);
                      fputs(ptr->RULE.BRANCH_set_5.OPR_2->
                            textval.s, fp);
                      fputs(" ", fp);
                      break;
                default:printf("ERROR : Wrong lexeme type
                      generated\n");
          }/* end switch */
          output_set(ptr->RULE.BRANCH_set_5.set_3);
          break;
    case 6:     /*It is alternative  no 6     */
          fputs("6 ", fp);
          output_set(ptr->RULE.BRANCH_set_6.set_1);
          switch(ptr->RULE.BRANCH_set_6.OPR_2->type)
          {
                case 1:     /*It is an operator     */
                      fputs("1 ", fp);
                      fputs(ptr->RULE.BRANCH_set_6.OPR_2->
                            textval.s, fp);
                      fputs(" ", fp);
                      break;
                default:printf("ERROR : Wrong lexeme type
                      generated\n");
          }/* end switch */
          output_typeid(ptr->RULE.BRANCH_set_6.typeid_3);
          break;
    case 7:     /*It is alternative  no 7     */
          fputs("7 ", fp);
          output_set(ptr->RULE.BRANCH_set_7.set_1);
          output_comp(ptr->RULE.BRANCH_set_7.comp_2);
          output_singleton(ptr->
                RULE.BRANCH_set_7.singleton_3);
          break;
    case 8:     /*It is alternative  no 8     */
          fputs("8 ", fp);
          output_set(ptr->RULE.BRANCH_set_8.set_1);
          output_comp(ptr->RULE.BRANCH_set_8.comp_2);
          output_quant(ptr->RULE.BRANCH_set_8.quant_3);
          output_set(ptr->RULE.BRANCH_set_8.set_4);
          break;
    case 9:     /*It is alternative  no 9     */
          fputs("9 ", fp);
          output_set(ptr->RULE.BRANCH_set_9.set_1);
          switch(ptr->RULE.BRANCH_set_9.OPR_2->type)
          {
                case 1:     /*It is an operator     */
                      fputs("1 ", fp);
                      fputs(ptr->RULE.BRANCH_set_9.OPR_2->
                            textval.s, fp);
                      fputs(" ", fp);
                      break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                    }/* end switch */
                    switch(ptr->RULE.BRANCH_set_9.OPR_3->type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->RULE.BRANCH_set_9.OPR_3->
                                      textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    output_pred(ptr->RULE.BRANCH_set_9.pred_4);
                    break;
               default:printf("ERROR: illegal alternative no.\n");
          }/* end switch */

#ifdef      DEBUG
printf("Out output_set\n");
#endif

}/* end output_set*/


/*routine to traverse a pred branch and output it out to file*/

output_pred(ptr)
PRED_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_pred\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_pred_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_pred_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_pred_1.singleton_2);
                  output_pred(ptr->RULE.BRANCH_pred_1.pred_3);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_singleton(ptr->
                        RULE.BRANCH_pred_2.singleton_1);
                  break;
            case 3:     /*It is alternative  no 3     */
```

```
                        fputs("3 ", fp);
                        switch(ptr->RULE.BRANCH_pred_3.OPR_1->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_pred_3.OPR_1->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_quant(ptr->RULE.BRANCH_pred_3.quant_2);
                        output_set(ptr->RULE.BRANCH_pred_3.set_3);
                        output_pred(ptr->RULE.BRANCH_pred_3.pred_4);
                        break;
                  case 4:     /*It is alternative  no 4     */
                        fputs("4 ", fp);
                        output_singleton(ptr->
                              RULE.BRANCH_pred_4.singleton_1);
                        output_comp(ptr->RULE.BRANCH_pred_4.comp_2);
                        output_singleton(ptr->
                              RULE.BRANCH_pred_4.singleton_3);
                        break;
                  case 5:     /*It is alternative  no 5     */
                        fputs("5 ", fp);
                        output_singleton(ptr->
                              RULE.BRANCH_pred_5.singleton_1);
                        output_comp(ptr->RULE.BRANCH_pred_5.comp_2);
                        output_quant(ptr->RULE.BRANCH_pred_5.quant_3);
                        output_set(ptr->RULE.BRANCH_pred_5.set_4);
                        break;
                  case 6:     /*It is alternative  no 6     */
                        fputs("6 ", fp);
                        output_quant(ptr->RULE.BRANCH_pred_6.quant_1);
                        output_set(ptr->RULE.BRANCH_pred_6.set_2);
                        output_comp(ptr->RULE.BRANCH_pred_6.comp_3);
                        output_quant(ptr->RULE.BRANCH_pred_6.quant_4);
                        output_set(ptr->RULE.BRANCH_pred_6.set_5);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_pred\n");
#endif

}/* end output_pred*/



/*routine to traverse a singleton branch and output it out to
file*/

output_singleton(ptr)
SINGLETON_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_singleton\n");
#endif
```

```
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    output_expl(ptr->RULE.BRANCH_singleton_1.expl_1);
                    break;
              case 2:     /*It is alternative  no 2     */
                    fputs("2 ", fp);
                    output_expl(ptr->RULE.BRANCH_singleton_2.expl_1);
                    output_orexpl(ptr->
                          RULE.BRANCH_singleton_2.orexpl_2);
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_singleton\n");
#endif

}/* end output_singleton*/



/*routine to traverse a orexpl branch and output it out to file*/

output_orexpl(ptr)
OREXPl_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_orexpl\n");
#endif

        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    fputs("1 ", fp);
                    switch(ptr->RULE.BRANCH_orexpl_1.OPR_1->type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                      RULE.BRANCH_orexpl_1.OPR_1->
                                            textval.s, fp);
                                fputs(" ", fp);
                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    output_expl(ptr->RULE.BRANCH_orexpl_1.expl_2);
                    break;
              case 2:     /*It is alternative  no 2     */
                    fputs("2 ", fp);
                    output_orexpl(ptr-
>RULE.BRANCH_orexpl_2.orexpl_1);
                    switch(ptr->RULE.BRANCH_orexpl_2.OPR_2->type)
                    {
                          case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                      RULE.BRANCH_orexpl_2.OPR_2->
                                            textval.s, fp);
                                fputs(" ", fp);
```

```
                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                output_expl(ptr->RULE.BRANCH_orexpl_2.expl_3);
                break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_orexpl\n");
#endif

}/* end output_orexpl*/


/*routine to traverse a expl branch and output it out to file*/

output_expl(ptr)
EXPl_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_expl\n");
#endif

        switch(ptr->type)
        {
            case 1:     /*It is alternative  no 1    */
                fputs("1 ", fp);
                output_exp2(ptr->RULE.BRANCH_expl_1.exp2_1);
                break;
            case 2:     /*It is alternative  no 2    */
                fputs("2 ", fp);
                output_exp2(ptr->RULE.BRANCH_expl_2.exp2_1);
                output_andexp2(ptr->
                    RULE.BRANCH_expl_2.andexp2_2);
                break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_expl\n");
#endif

}/* end output_expl*/


/*routine to traverse a andexp2 branch and output it out to file*/

output_andexp2(ptr)
ANDEXP2_TYPE      ptr;
{

#ifdef      DEBUG
printf("In output_andexp2\n");
#endif

        switch(ptr->type)
        {
            case 1:     /*It is alternative  no 1    */
```

```
                        fputs("1 ", fp);
                        switch(ptr->RULE.BRANCH_andexp2_1.OPR_1->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_andexp2_1.OPR_1->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_exp2(ptr->RULE.BRANCH_andexp2_1.exp2_2);
                        break;
                  case 2:     /*It is alternative  no 2     */
                        fputs("2 ", fp);
                        output_andexp2(ptr->
                              RULE.BRANCH_andexp2_2.andexp2_1);
                        switch(ptr->RULE.BRANCH_andexp2_2.OPR_2->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_andexp2_2.OPR_2->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_exp2(ptr->RULE.BRANCH_andexp2_2.exp2_3);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_andexp2\n");
#endif

}/* end output_andexp2*/



/*routine to traverse a exp2 branch and output it out to file*/

output_exp2(ptr)
EXP2_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_exp2\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_exp3(ptr->RULE.BRANCH_exp2_1.exp3_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  switch(ptr->RULE.BRANCH_exp2_2.OPR_1->type)
```

```
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_exp2_2.OPR_1->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        output_exp3(ptr->RULE.BRANCH_exp2_2.exp3_2);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_exp2\n");
#endif

}/* end output_exp2*/



/*routine to traverse a exp3 branch and output it out to file*/

output_exp3(ptr)
EXP3_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_exp3\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_exp4(ptr->RULE.BRANCH_exp3_1.exp4_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_exp4(ptr->RULE.BRANCH_exp3_2.exp4_1);
                  output_comp(ptr->RULE.BRANCH_exp3_2.comp_2);
                  output_exp4(ptr->RULE.BRANCH_exp3_2.exp4_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_exp3\n");
#endif

}/* end output_exp3*/



/*routine to traverse a exp4 branch and output it out to file*/

output_exp4(ptr)
EXP4_TYPE   ptr;
{
```

```
#ifdef      DEBUG
printf("In output_exp4\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_exp5(ptr->RULE.BRANCH_exp4_1.exp5_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_exp5(ptr->RULE.BRANCH_exp4_2.exp5_1);
                  output_addop(ptr->RULE.BRANCH_exp4_2.addop_2);
                  output_exp5(ptr->RULE.BRANCH_exp4_2.exp5_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_exp4\n");
#endif

}/* end output_exp4*/



/*routine to traverse a exp5 branch and output it out to file*/

output_exp5(ptr)
EXP5_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_exp5\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  output_exp6(ptr->RULE.BRANCH_exp5_1.exp6_1);
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  output_exp6(ptr->RULE.BRANCH_exp5_2.exp6_1);
                  output_mulop(ptr->RULE.BRANCH_exp5_2.mulop_2);
                  output_exp6(ptr->RULE.BRANCH_exp5_2.exp6_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_exp5\n");
#endif

}/* end output_exp5*/



/*routine to traverse a exp6 branch and output it out to file*/

output_exp6(ptr)
```

```
EXP6_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_exp6\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  fputs("1 ", fp);
                  output_exp7(ptr->RULE.BRANCH_exp6_1.exp7_1);
                  break;
            case 2:     /*It is alternative  no 2    */
                  fputs("2 ", fp);
                  output_exp7(ptr->RULE.BRANCH_exp6_2.exp7_1);
                  switch(ptr->RULE.BRANCH_exp6_2.OPR_2->type)
                  {
                        case 1:     /*It is an operator    */
                              fputs("1 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_exp6_2.OPR_2->
                                    textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_typeid(ptr->RULE.BRANCH_exp6_2.typeid_3);
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_exp6\n");
#endif

}/* end output_exp6*/



/*routine to traverse a exp7 branch and output it out to file*/

output_exp7(ptr)
EXP7_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_exp7\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  fputs("1 ", fp);
                  output_constant(ptr->
                        RULE.BRANCH_exp7_1.constant_1);
                  break;
            case 2:     /*It is alternative  no 2    */
                  fputs("2 ", fp);
                  output_vblid(ptr->RULE.BRANCH_exp7_2.vblid_1);
                  break;
            case 3:     /*It is alternative  no 3    */
```

```
            fputs("3 ", fp);
            output_mvfuncall(ptr->
                  RULE.BRANCH_exp7_3.mvfuncall_1);
            break;
      case 4:     /*It is alternative  no 4     */
            fputs("4 ", fp);
            output_aggcall(ptr->
                  RULE.BRANCH_exp7_4.aggcall_1);
            break;
      case 5:     /*It is alternative  no 5     */
            fputs("5 ", fp);
            output_quant(ptr->RULE.BRANCH_exp7_5.quant_1);
            output_set(ptr->RULE.BRANCH_exp7_5.set_2);
            switch(ptr->RULE.BRANCH_exp7_5.OPR_3->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_exp7_5.OPR_3->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
                  RULE.BRANCH_exp7_5.singleton_4);
            break;
      case 6:     /*It is alternative  no 6     */
            fputs("6 ", fp);
            switch(ptr->RULE.BRANCH_exp7_6.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_exp7_6.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_set(ptr->RULE.BRANCH_exp7_6.set_2);
            break;
      case 7:     /*It is alternative  no 7     */
            fputs("7 ", fp);
            switch(ptr->RULE.BRANCH_exp7_7.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_exp7_7.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
                  RULE.BRANCH_exp7_7.singleton_2);
            switch(ptr->RULE.BRANCH_exp7_7.OPR_3->type)
            {
                  case 1:     /*It is an operator     */
```

```
                                    fputs("1 ", fp);
                                    fputs(ptr->RULE.BRANCH_exp7_7.OPR_3->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_exp7\n");
#endif

}/* end output_exp7*/


/*routine to traverse a aggcall branch and output it out to file*/

output_aggcall(ptr)
AGGCALL_TYPE      ptr;
{

#ifdef      DEBUG
printf("In output_aggcall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_aggcall_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_aggcall_1.OPR_2->
                                    textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_set(ptr->RULE.BRANCH_aggcall_1.set_3);
                  switch(ptr->RULE.BRANCH_aggcall_1.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
```

```
                        fputs(ptr->
                              RULE.BRANCH_aggcall_1.OPR_4->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 2:     /*It is alternative  no 2     */
            fputs("2 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_2.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_2.OPR_2->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_2.OPR_2->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_set(ptr->RULE.BRANCH_aggcall_2.set_3);
            switch(ptr->RULE.BRANCH_aggcall_2.OPR_4->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_2.OPR_4->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 3:     /*It is alternative  no 3     */
            fputs("3 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_3.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_3.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_3.OPR_2->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
```

```
                        fputs(ptr->
                              RULE.BRANCH_aggcall_3.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_set(ptr->RULE.BRANCH_aggcall_3.set_3);
            switch(ptr->RULE.BRANCH_aggcall_3.OPR_4->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_3.OPR_4->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 4:     /*It is alternative  no 4     */
            fputs("4 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_4.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_4.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_4.OPR_2->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_4.OPR_2->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
                  RULE.BRANCH_aggcall_4.singleton_3);
            switch(ptr->RULE.BRANCH_aggcall_4.OPR_4->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_4.OPR_4->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_mtuple(ptr->
```

```
                RULE.BRANCH_aggcall_4.mtuple_5);
            switch(ptr->RULE.BRANCH_aggcall_4.OPR_6->type)
            {
                case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                                RULE.BRANCH_aggcall_4.OPR_6->
                                        textval.s, fp);
                        fputs(" ", fp);
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
        case 5:     /*It is alternative  no 5     */
            fputs("5 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_5.OPR_1->type)
            {
                case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                                RULE.BRANCH_aggcall_5.OPR_1->
                                        textval.s, fp);
                        fputs(" ", fp);
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_5.OPR_2->type)
            {
                case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                                RULE.BRANCH_aggcall_5.OPR_2->
                                        textval.s, fp);
                        fputs(" ", fp);
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
                RULE.BRANCH_aggcall_5.singleton_3);
            switch(ptr->RULE.BRANCH_aggcall_5.OPR_4->type)
            {
                case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                                RULE.BRANCH_aggcall_5.OPR_4->
                                        textval.s, fp);
                        fputs(" ", fp);
                        break;
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_mtuple(ptr->
                RULE.BRANCH_aggcall_5.mtuple_5);
            switch(ptr->RULE.BRANCH_aggcall_5.OPR_6->type)
            {
                case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                                RULE.BRANCH_aggcall_5.OPR_6->
                                        textval.s, fp);
```

```
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 6:     /*It is alternative  no 6     */
            fputs("6 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_6.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_6.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_6.OPR_2->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_6.OPR_2->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
                  RULE.BRANCH_aggcall_6.singleton_3);
            switch(ptr->RULE.BRANCH_aggcall_6.OPR_4->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_6.OPR_4->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 7:     /*It is alternative  no 7     */
            fputs("7 ", fp);
            switch(ptr->RULE.BRANCH_aggcall_7.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_aggcall_7.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            switch(ptr->RULE.BRANCH_aggcall_7.OPR_2->type)
            {
```

```
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_aggcall_7.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_singleton(ptr->
                        RULE.BRANCH_aggcall_7.singleton_3);
                  switch(ptr->RULE.BRANCH_aggcall_7.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_aggcall_7.OPR_4-
>textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_mtuple(ptr->
                        RULE.BRANCH_aggcall_7.mtuple_5);
                  switch(ptr->RULE.BRANCH_aggcall_7.OPR_6->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_aggcall_7.OPR_6->
                                    textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_aggcall\n");
#endif

}/* end output_aggcall*/



/*routine to traverse a comp branch and output it out to file*/

output_comp(ptr)
COMP_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_comp\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
```

```
        fputs("1 ", fp);
        switch(ptr->RULE.BRANCH_comp_1.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->RULE.BRANCH_comp_1.OPR_1->
                          textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 2:     /*It is alternative  no 2     */
        fputs("2 ", fp);
        switch(ptr->RULE.BRANCH_comp_2.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                          RULE.BRANCH_comp_2.OPR_1->
                                textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 3:     /*It is alternative  no 3     */
        fputs("3 ", fp);
        switch(ptr->RULE.BRANCH_comp_3.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr->
                          RULE.BRANCH_comp_3.OPR_1->
                                textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 4:     /*It is alternative  no 4     */
        fputs("4 ", fp);
        switch(ptr->RULE.BRANCH_comp_4.OPR_1->type)
        {
              case 1:     /*It is an operator     */
                    fputs("1 ", fp);
                    fputs(ptr-
                          RULE.BRANCH_comp_4.OPR_1->
                                textval.s, fp);
                    fputs(" ", fp);
                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 5:     /*It is alternative  no 5     */
        fputs("5 ", fp);
        switch(ptr->RULE.BRANCH_comp_5.OPR_1->type)
        {
              case 1:     /*It is an operator     */
```

```
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_comp_5.OPR_1->
                                                textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  case 6:     /*It is alternative  no 6     */
                        fputs("6 ", fp);
                        switch(ptr->RULE.BRANCH_comp_6.OPR_1->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->RULE.BRANCH_comp_6.OPR_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_comp\n");
#endif

}/* end output_comp*/



/*routine to traverse a quant branch and output it out to file*/

output_quant(ptr)
QUANT_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_quant\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_quant_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_quant_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
```

```
            fputs("2 ", fp);
            switch(ptr->RULE.BRANCH_quant_2.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_quant_2.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 3:     /*It is alternative  no 3     */
            fputs("3 ", fp);
            switch(ptr->RULE.BRANCH_quant_3.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_quant_3.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 4:     /*It is alternative  no 4     */
            fputs("4 ", fp);
            switch(ptr->RULE.BRANCH_quant_4.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_quant_4.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_leastmost(ptr->
                  RULE.BRANCH_quant_4.leastmost_2);
            output_singleton(ptr->
                  RULE.BRANCH_quant_4.singleton_3);
            break;
      case 5:     /*It is alternative  no 5     */
            fputs("5 ", fp);
            switch(ptr->RULE.BRANCH_quant_5.OPR_1->type)
            {
                  case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_quant_5.OPR_1->
                                    textval.s, fp);
                        fputs(" ", fp);
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            output_singleton(ptr->
```

```
                        RULE.BRANCH_quant_5.singleton_2);
                    break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_quant\n");
#endif

}/* end output_quant*/



/*routine to traverse a leastmost branch and output it out to
file*/

output_leastmost(ptr)
LEASTMOST_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_leastmost\n");
#endif

        switch(ptr->type)
        {
            case 1:     /*It is alternative  no 1     */
                fputs("1 ", fp);
                switch(ptr->RULE.BRANCH_leastmost_1.OPR_1->type)
                {
                    case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                            RULE.BRANCH_leastmost_1.OPR_1->
                                textval.s, fp);
                        fputs(" ", fp);
                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
            case 2:     /*It is alternative  no 2     */
                fputs("2 ", fp);
                switch(ptr->RULE.BRANCH_leastmost_2.OPR_1->type)
                {
                    case 1:     /*It is an operator     */
                        fputs("1 ", fp);
                        fputs(ptr->
                            RULE.BRANCH_leastmost_2.OPR_1->
                                textval.s, fp);
                        fputs(" ", fp);
                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out output_leastmost\n");
#endif
```

```
}/* end output_leastmost*/


/*routine to traverse a addop branch and output it out to file*/

output_addop(ptr)
ADDOP_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_addop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_addop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_addop_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  switch(ptr->RULE.BRANCH_addop_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_addop_2.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_addop\n");
#endif

}/* end output_addop*/


/*routine to traverse a mulop branch and output it out to file*/

output_mulop(ptr)
MULOP_TYPE  ptr;
{
```

```
#ifdef      DEBUG
printf("In output_mulop\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_mulop_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_mulop_1.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  fputs("2 ", fp);
                  switch(ptr->RULE.BRANCH_mulop_2.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_mulop_2.OPR_1->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_mulop\n");
#endif

}/* end output_mulop*/



/*routine to traverse a constant branch and output it out to
file*/

output_constant(ptr)
CONSTANT_TYPE     ptr;
{

#ifdef      DEBUG
printf("In output_constant\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
```

```
                        switch(ptr->RULE.BRANCH_constant_1.INTEGER_1->
                              type)
                        {
                              case 3:     /*It is an integer      */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                          RULE.BRANCH_constant_1.
                                          INTEGER_1->textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  case 2:     /*It is alternative  no 2     */
                        fputs("2 ", fp);
                        switch(ptr->RULE.BRANCH_constant_2.STRING_1->
                              type)
                        {
                              case 4:     /*It is a string        */
                                    fputs("1 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_constant_2.STRING_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  case 3:     /*It is alternative  no 3     */
                        fputs("3 ", fp);
                        output_bool(ptr->RULE.BRANCH_constant_3.bool_1);
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_constant\n");
#endif

}/* end output_constant*/



/*routine to traverse a bool branch and output it out to file*/

output_bool(ptr)
BOOL_TYPE   ptr;
{

#ifdef      DEBUG
printf("In output_bool\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_bool_1.OPR_1->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
```

```
                                    fputs(ptr->RULE.BRANCH_bool_1.OPR_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  case 2:     /*It is alternative  no 2     */
                        fputs("2 ", fp);
                        switch(ptr->RULE.BRANCH_bool_2.OPR_1->type)
                        {
                              case 1:     /*It is an operator     */
                                    fputs("1 ", fp);
                                    fputs(ptr->RULE.BRANCH_bool_2.
                                          OPR_1->textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_bool\n");
#endif

}/* end output_bool*/



/*routine to traverse a funcid branch and output it out to file*/

output_funcid(ptr)
FUNCID_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_funcid\n");
#endif

            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        fputs("1 ", fp);
                        switch(ptr->RULE.BRANCH_funcid_1.IDENTIFIERF_1->
                              type)
                        {
                              case 6:     /*It is an identifier   */
                                    fputs("6 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_funcid_1.IDENTIFIERF_1->
                                          textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */
```

```
#ifdef      DEBUG
printf("Out output_funcid\n");
#endif

}/* end output_funcid*/


/*routine to traverse a typeid branch and output it out to file*/

output_typeid(ptr)
TYPEID_TYPE       ptr;
{

#ifdef      DEBUG
printf("In output_typeid\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->RULE.BRANCH_typeid_1.IDENTIFIER_1->
                        type)
                  {
                        case 2:     /*It is an identifier   */
                              fputs("2 ", fp);
                        fputs(ptr->
                              RULE.BRANCH_typeid_1.IDENTIFIER_1->
                                    textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_typeid\n");
#endif

}/* end output_typeid*/


/*routine to traverse a vblid branch and output it out to file*/

output_vblid(ptr)
VBLID_TYPE  ptr;
{

#ifdef      DEBUG
printf("In output_vblid\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
```

```
                        switch(ptr->RULE.BRANCH_vblid_1.IDENTIFIERV_1->
                              type)
                        {
                              case 5:     /*It is an var identifier    */
                                    fputs("5 ", fp);
                                    fputs(ptr->
                                    RULE.BRANCH_vblid_1.
                                          IDENTIFIERV_1->textval.s, fp);
                                    fputs(" ", fp);
                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    enerated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_vblid\n");
#endif

}/* end output_vblid*/



/*routine to traverse a svfuncall branch and output it out to
file*/

output_svfuncall(ptr)
SVFUNCALL_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_svfuncall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  fputs("1 ", fp);
                  switch(ptr->
                        RULE.BRANCH_svfuncall_1.IDENTIFIERF_1->
                              type)
                  {
                        case 6:     /*It is an identifier  */
                              fputs("6 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_svfuncall_1.
                                    IDENTIFIERF_1->textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_svfuncall_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator    */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_svfuncall_1.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
```

```
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                output_singleton(ptr->
                        RULE.BRANCH_svfuncall_1.singleton_3);
                switch(ptr->RULE.BRANCH_svfuncall_1.OPR_4->type)
                {
                        case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                        RULE.BRANCH_svfuncall_1.OPR_4->
                                                textval.s, fp);
                                fputs(" ", fp);
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        case 2:     /*It is alternative  no 2     */
                fputs("2 ", fp);
                switch(ptr->
                        RULE.BRANCH_svfuncall_2.IDENTIFIERF_1->
                                type)
                {
                        case 6:     /*It is an identifier   */
                                fputs("6 ", fp);
                                fputs(ptr->
                                        RULE.BRANCH_svfuncall_2.
                                        IDENTIFIERF_1->textval.s, fp);
                                fputs(" ", fp);
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                switch(ptr->RULE.BRANCH_svfuncall_2.OPR_2->type)
                {
                        case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                        RULE.BRANCH_svfuncall_2.OPR_2->
                                                textval.s, fp);
                                fputs(" ", fp);
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                output_singleton(ptr->
                        RULE.BRANCH_svfuncall_2.singleton_3);
                output_singlist(ptr->
                        RULE.BRANCH_svfuncall_2.singlist_4);
                switch(ptr->RULE.BRANCH_svfuncall_2.OPR_5->type)
                {
                        case 1:     /*It is an operator     */
                                fputs("1 ", fp);
                                fputs(ptr->
                                RULE.BRANCH_svfuncall_2.OPR_5->
                                        textval.s, fp);
                                fputs(" ", fp);
                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
```

```
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out output_svfuncall\n");
#endif

}/* end output_svfuncall*/



/*routine to traverse a mvfuncall branch and output it out to
file*/

output_mvfuncall(ptr)
MVFUNCALL_TYPE    ptr;
{

#ifdef      DEBUG
printf("In output_mvfuncall\n");
#endif

      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  fputs("1 ", fp);
                  switch(ptr->
                        RULE.BRANCH_mvfuncall_1.IDENTIFIERF_1->
                              type)
                  {
                        case 6:     /*It is an identifier   */
                              fputs("6 ", fp);
                              fputs(ptr->
                              RULE.BRANCH_mvfuncall_1.
                                    IDENTIFIERF_1->textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_2->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_mvfuncall_1.OPR_2->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  output_mtuple(ptr->
                        RULE.BRANCH_mvfuncall_1.mtuple_3);
                  switch(ptr->RULE.BRANCH_mvfuncall_1.OPR_4->type)
                  {
                        case 1:     /*It is an operator     */
                              fputs("1 ", fp);
                              fputs(ptr->
                                    RULE.BRANCH_mvfuncall_1.OPR_4->
                                          textval.s, fp);
                              fputs(" ", fp);
                              break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
                default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out output_mvfuncall\n");
#endif

}/* end output_mvfuncall*/
```

```
/*    FILE: input.c     */

/* This file contains the routines to input branches of the syntax
tree*/

/*    includes for this file */

#include          <stdio.h>
#include          "structures.h"

/*    externals for this file      */

extern            FILE *fp;
extern            output_mtuple();
extern            output_definetypes();

/*    locals for this file    */

char              name[25];
int               *tptr;
int               temptype;



input_program()
{
      PROGRAM_TYPE      ptr;
      int         *ptr1;

#ifdef      DEBUG
printf("In input_program\n");
#endif

      ptr = (PROGRAM_TYPE)malloc(sizeof(struct program_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_statements();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_program\n");
#endif

      return((int) ptr);

}/* end input_program*/



input_statements()
{
STATEMENTS_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
```

```
printf("In input_statements\n");
#endif

      ptr = (STATEMENTS_TYPE)malloc(sizeof(struct
            statements_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_statement();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_statements();
                  ptr1++;
                  *ptr1 = input_statement();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_statements\n");
#endif

      return((int) ptr);

}/* end input_statements*/



input_statement()
{
STATEMENT_TYPE    ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_statement\n");
#endif

      ptr = (STATEMENT_TYPE)malloc(sizeof(struct statement_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_declarative();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_imperative();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_statement\n");
#endif

      return((int) ptr);
```

```
}/* end input_statement*/


input_declarative()
{
DECLARATIVE_TYPE  ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_declarative\n");
#endif

      ptr = (DECLARATIVE_TYPE)malloc(sizeof(struct
            declarative_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_simple_decl();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_complex_decl();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_declarative\n");
#endif

      return((int) ptr);

}/* end input_declarative*/


input_simple_decl()
{
SIMPLE_DECL_TYPE  ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_simple_decl\n");
#endif

      ptr = (SIMPLE_DECL_TYPE)malloc(sizeof(struct
            simple_decl_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
```

```
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = input_funcid();
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = input_arrow();
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 2:     /*It is alternative  no 2     */
```

```
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_funcid();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_vblid();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_arrow();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
```

```
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
case 3:     /*It is alternative  no 3    */
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator    */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_funcid();
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator    */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_vblid();
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator    */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_arrow();
```

```
                        ptrl++;
                        *ptrl = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptrl)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptrl);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_simple_decl\n");
#endif

            return((int) ptr);
}/* end input_simple_decl*/



input_complex_decl()
{
COMPLEX_DECL_TYPE       ptr;
int *ptrl;

#ifdef      DEBUG
printf("In input_complex_decl\n");
#endif

            ptr = (COMPLEX_DECL_TYPE)malloc(sizeof(struct
                  complex_decl_type));
            ptrl = (int *)ptr;
            fscanf(fp, "%d", ptr);
            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        ptrl++;
                        *ptrl = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptrl)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptrl);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptrl++;
                        *ptrl = input_funcspec();
```

```
                        ptrl++;
                        *ptrl = input_arrow();
                        ptrl++;
                        *ptrl = input_definetypes();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_complex_decl\n");
#endif

            return((int) ptr);

}/* end input_complex_decl*/



input_definetypes()
{
DEFINETYPES_TYPE  ptr;
int *ptrl;

#ifdef      DEBUG
printf("In input_definetypes\n");
#endif

            ptr = (DEFINETYPES_TYPE)malloc(sizeof(struct
                  definetypes_type));
            ptrl = (int *)ptr;
            fscanf(fp, "%d", ptr);
            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        ptrl++;
                        *ptrl = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptrl)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptrl);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptrl++;
                        *ptrl = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptrl)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptrl);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
```

```
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = input_funcspec();
                break;
        case 2:     /*It is alternative  no 2     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = input_expr();
                break;
        case 3:     /*It is alternative  no 3     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
```

```
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_mtuple();
        break;
case 4:     /*It is alternative  no 4     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_mtuple();
        break;
case 5:     /*It is alternative  no 5     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);
```

```
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_mtuple();
            break;
      case 6:     /*It is alternative  no 6     */
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_mtuple();
            break;
      case 7:     /*It is alternative  no 7     */
            ptrl++;
            *ptrl = input_expr();
```

```
                    break;
               default:printf("ERROR: illegal alternative no.\n");
          }/* end switch */

#ifdef      DEBUG
printf("Out input_definetypes\n");
#endif

          return((int) ptr);

}/* end input_definetypes*/



input_arrow()
{
ARROW_TYPE  ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_arrow\n");
#endif

          ptr = (ARROW_TYPE)malloc(sizeof(struct arrow_type));
          ptr1 = (int *)ptr;
          fscanf(fp, "%d", ptr);
          switch(ptr->type)
          {
               case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                         case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                         default:printf("ERROR : Wrong lexeme type
                              generated\n");
                    }/* end switch */
                    break;
               case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                         case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                         default:printf("ERROR : Wrong lexeme type
                              generated\n");
                    }/* end switch */
```

```
                    break;
               default:printf("ERROR: illegal alternative no.\n");
          }/* end switch */

#ifdef      DEBUG
printf("Out input_arrow\n");
#endif

          return((int) ptr);

}/* end input_arrow*/



input_funcspec()
{
FUNCSPEC_TYPE     ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_funcspec\n");
#endif

          ptr = (FUNCSPEC_TYPE)malloc(sizeof(struct funcspec_type));
          ptr1 = (int *)ptr;
          fscanf(fp, "%d", ptr);
          switch(ptr->type)
          {
               case 1:     /*It is alternative  no 1     */
                         ptr1++;
                         *ptr1 = input_funcid();
                         ptr1++;
                         *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                         fscanf(fp, "%d", &temptype);
                         ((OPR_TYPE *) *ptr1)->type = temptype;
                         switch(temptype)
                         {
                              case 1:     /*It is an operator     */
                                   tptr = (int *) (*ptr1);
                                   fscanf(fp, "%s", name);
                                   *tptr = (int )malloc(strlen(name)+1);
                                   strcpy((char *) (*tptr), name);

                                   break;
                              default:printf("ERROR : Wrong lexeme type
                                   generated\n");
                         }/* end switch */
                         ptr1++;
                         *ptr1 = input_arglist();
                         ptr1++;
                         *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                         fscanf(fp, "%d", &temptype);
                         ((OPR_TYPE *) *ptr1)->type = temptype;
                         switch(temptype)
                         {
                              case 1:     /*It is an operator     */
                                   tptr = (int *) (*ptr1);
                                   fscanf(fp, "%s", name);
                                   *tptr = (int )malloc(strlen(name)+1);
                                   strcpy((char *) (*tptr), name);

                                   break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
                default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_funcspec\n");
#endif

        return((int) ptr);

}/* end input_funcspec*/



input_arglist()
{
ARGLIST_TYPE      ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_arglist\n");
#endif

        ptr = (ARGLIST_TYPE)malloc(sizeof(struct arglist_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = input_typeid();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_arglist();
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    ptr1++;
                    *ptr1 = input_typeid();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_arglist\n");
#endif
```

```
        return((int) ptr);

}/* end input_arglist*/



input_mtuple()
{
MTUPLE_TYPE       ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_mtuple\n");
#endif

        ptr = (MTUPLE_TYPE)malloc(sizeof(struct mtuple_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = input_expr();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_mtuple();
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    ptr1++;
                    *ptr1 = input_expr();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_mtuple\n");
#endif

        return((int) ptr);

}/* end input_mtuple*/



input_stuple()
{
STUPLE_TYPE       ptr;
```

```
int *ptr1;

#ifdef      DEBUG
printf("In input_stuple\n");
#endif

      ptr = (STUPLE_TYPE)malloc(sizeof(struct stuple_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_singleton();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_stuple();
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_singleton();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_stuple\n");
#endif

      return((int) ptr);

}/* end input_stuple*/



input_singlist()
{
SINGLIST_TYPE     ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_singlist\n");
#endif

      ptr = (SINGLIST_TYPE)malloc(sizeof(struct singlist_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
```

```
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    ptr1++;
                    *ptr1 = input_singleton();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_singlist();
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    ptr1++;
                    *ptr1 = input_singleton();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_singlist\n");
#endif

        return((int) ptr);

}/* end input_singlist*/



input_expr()
{
EXPR_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_expr\n");
```

```
#endif

      ptr = (EXPR_TYPE)malloc(sizeof(struct expr_type));
      ptrl = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptrl++;
                  *ptrl = input_set();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptrl++;
                  *ptrl = input_singleton();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_expr\n");
#endif

      return((int) ptr);

}/* end input_expr*/



input_imperative()
{
IMPERATIVE_TYPE   ptr;
int *ptrl;

#ifdef      DEBUG
printf("In input_imperative\n");
#endif

      ptr = (IMPERATIVE_TYPE)malloc(sizeof(struct
            imperative_type));
      ptrl = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptrl++;
                  *ptrl = input_forloop();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptrl++;
                  *ptrl = input_gpimperative();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_imperative\n");
#endif

      return((int) ptr);

}/* end input_imperative*/
```

```
input_forloop()
{
FORLOOP_TYPE      ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_forloop\n");
#endif

      ptr = (FORLOOP_TYPE)malloc(sizeof(struct forloop_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_set();
                  ptr1++;
                  *ptr1 = input_imperative();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
```

```
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptr1++;
                        *ptr1 = input_singleton();
                        ptr1++;
                        *ptr1 = input_imperative();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_forloop\n");
#endif

            return((int) ptr);

}/* end input_forloop*/



input_gpimperative()
{
GPIMPERATIVE_TYPE       ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_gpimperative\n");
#endif

            ptr = (GPIMPERATIVE_TYPE)malloc(sizeof(struct
                  gpimperative_type));
            ptr1 = (int *)ptr;
            fscanf(fp, "%d", ptr);
            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        ptr1++;
                        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptr1)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptr1);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptr1++;
                        *ptr1 = input_stuple();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */
```

```
#ifdef      DEBUG
printf("Out input_gpimperative\n");
#endif

      return((int) ptr);

}/* end input_gpimperative*/



input_set()
{
SET_TYPE    ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_set\n");
#endif

      ptr = (SET_TYPE)malloc(sizeof(struct set_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1    */
                  ptr1++;
                  *ptr1 = input_typeid();
                  break;
            case 2:     /*It is alternative  no 2    */
                  ptr1++;
                  *ptr1 = input_mvfuncall();
                  break;
            case 3:     /*It is alternative  no 3    */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_stuple();
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);
```

```
                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
case 4:     /*It is alternative  no 4     */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = input_set();
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
case 5:     /*It is alternative  no 5     */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(IDENTIFIER_TYPE));
            fscanf(fp, "%d", &temptype);
            ((IDENTIFIER_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 2:     /*It is an identifier   */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
```

```
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_set();
        break;
  case 6:     /*It is alternative  no 6     */
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_typeid();
        break;
  case 7:     /*It is alternative  no 7     */
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = input_comp();
        ptr1++;
        *ptr1 = input_singleton();
        break;
  case 8:     /*It is alternative  no 8     */
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = input_comp();
        ptr1++;
        *ptr1 = input_quant();
        ptr1++;
        *ptr1 = input_set();
        break;
  case 9:     /*It is alternative  no 9     */
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
```

```
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_pred();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_set\n");
#endif

      return((int) ptr);

}/* end input_set*/



input_pred()
{
PRED_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_pred\n");
#endif

      ptr = (PRED_TYPE)malloc(sizeof(struct pred_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {

            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
```

```
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_singleton();
        ptr1++;
        *ptr1 = input_pred();
        break;
case 2:     /*It is alternative  no 2     */
        ptr1++;
        *ptr1 = input_singleton();
        break;
case 3:     /*It is alternative  no 3     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_quant();
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = input_pred();
        break;
case 4:     /*It is alternative  no 4     */
        ptr1++;
        *ptr1 = input_singleton();
        ptr1++;
        *ptr1 = input_comp();
        ptr1++;
        *ptr1 = input_singleton();
        break;
case 5:     /*It is alternative  no 5     */
        ptr1++;
        *ptr1 = input_singleton();
        ptr1++;
        *ptr1 = input_comp();
        ptr1++;
        *ptr1 = input_quant();
        ptr1++;
        *ptr1 = input_set();
        break;
case 6:     /*It is alternative  no 6     */
```

```
                    ptr1++;
                    *ptr1 = input_quant();
                    ptr1++;
                    *ptr1 = input_set();
                    ptr1++;
                    *ptr1 = input_comp();
                    ptr1++;
                    *ptr1 = input_quant();
                    ptr1++;
                    *ptr1 = input_set();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_pred\n");
#endif

        return((int) ptr);

}/* end input_pred*/



input_singleton()
{
SINGLETON_TYPE    ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_singleton\n");
#endif

        ptr = (SINGLETON_TYPE)malloc(sizeof(struct singleton_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = input_expl();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_expl();
                    ptr1++;
                    *ptr1 = input_orexpl();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_singleton\n");
#endif

        return((int) ptr);

}/* end input_singleton*/



input_orexpl()
{
```

```
OREXP1_TYPE       ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_orexpl\n");
#endif

      ptr = (OREXP1_TYPE)malloc(sizeof(struct orexpl_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_expl();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_orexpl();
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_expl();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_orexpl\n");
#endif

      return((int) ptr);
```

```
}/* end input_orexp1*/


input_exp1()
{
EXP1_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp1\n");
#endif

      ptr = (EXP1_TYPE)malloc(sizeof(struct exp1_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_exp2();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_exp2();
                  ptr1++;
                  *ptr1 = input_andexp2();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_exp1\n");
#endif

      return((int) ptr);

}/* end input_exp1*/


input_andexp2()
{
ANDEXP2_TYPE      ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_andexp2\n");
#endif

      ptr = (ANDEXP2_TYPE)malloc(sizeof(struct andexp2_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
```

```
                                    tptr = (int *) (*ptr1);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptr1++;
                        *ptr1 = input_exp2();
                        break;
                  case 2:     /*It is alternative  no 2     */
                        ptr1++;
                        *ptr1 = input_andexp2();
                        ptr1++;
                        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptr1)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptr1);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptr1++;
                        *ptr1 = input_exp2();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_andexp2\n");
#endif

            return((int) ptr);

}/* end input_andexp2*/



input_exp2()
{
EXP2_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp2\n");
#endif

            ptr = (EXP2_TYPE)malloc(sizeof(struct exp2_type));
            ptr1 = (int *)ptr;
            fscanf(fp, "%d", ptr);
            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        ptr1++;
```

```
                        *ptr1 = input_exp3();
                        break;
                  case 2:     /*It is alternative  no 2     */
                        ptr1++;
                        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                        fscanf(fp, "%d", &temptype);
                        ((OPR_TYPE *) *ptr1)->type = temptype;
                        switch(temptype)
                        {
                              case 1:     /*It is an operator     */
                                    tptr = (int *) (*ptr1);
                                    fscanf(fp, "%s", name);
                                    *tptr = (int )malloc(strlen(name)+1);
                                    strcpy((char *) (*tptr), name);

                                    break;
                              default:printf("ERROR : Wrong lexeme type
                                    generated\n");
                        }/* end switch */
                        ptr1++;
                        *ptr1 = input_exp3();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_exp2\n");
#endif

            return((int) ptr);

}/* end input_exp2*/



input_exp3()
{
EXP3_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp3\n");
#endif

            ptr = (EXP3_TYPE)malloc(sizeof(struct exp3_type));
            ptr1 = (int *)ptr;
            fscanf(fp, "%d", ptr);
            switch(ptr->type)
            {
                  case 1:     /*It is alternative  no 1     */
                        ptr1++;
                        *ptr1 = input_exp4();
                        break;
                  case 2:     /*It is alternative  no 2     */
                        ptr1++;
                        *ptr1 = input_exp4();
                        ptr1++;
                        *ptr1 = input_comp();
                        ptr1++;
                        *ptr1 = input_exp4();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */
```

```
#ifdef      DEBUG
printf("Out input_exp3\n");
#endif

      return((int) ptr);

}/* end input_exp3*/



input_exp4()
{
EXP4_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp4\n");
#endif

      ptr = (EXP4_TYPE)malloc(sizeof(struct exp4_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_exp5();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_exp5();
                  ptr1++;
                  *ptr1 = input_addop();
                  ptr1++;
                  *ptr1 = input_exp5();
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_exp4\n");
#endif

      return((int) ptr);

}/* end input_exp4*/



input_exp5()
{
EXP5_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp5\n");
#endif

      ptr = (EXP5_TYPE)malloc(sizeof(struct exp5_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
```

```
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = input_exp6();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_exp6();
                    ptr1++;
                    *ptr1 = input_mulop();
                    ptr1++;
                    *ptr1 = input_exp6();
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_exp5\n");
#endif

        return((int) ptr);

}/* end input_exp5*/



input_exp6()
{
EXP6_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp6\n");
#endif

        ptr = (EXP6_TYPE)malloc(sizeof(struct exp6_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = input_exp7();
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = input_exp7();
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
```

```
                        ptr1++;
                        *ptr1 = input_typeid();
                        break;
                  default:printf("ERROR: illegal alternative no.\n");
            }/* end switch */

#ifdef      DEBUG
printf("Out input_exp6\n");
#endif

      return((int) ptr);

}/* end input_exp6*/



input_exp7()
{
EXP5_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_exp7\n");
#endif

      ptr = (EXP5_TYPE)malloc(sizeof(struct exp7_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = input_constant();
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = input_vblid();
                  break;
            case 3:     /*It is alternative  no 3     */
                  ptr1++;
                  *ptr1 = input_mvfuncall();
                  break;
            case 4:     /*It is alternative  no 4     */
                  ptr1++;
                  *ptr1 = input_aggcall();
                  break;
            case 5:     /*It is alternative  no 5     */
                  ptr1++;
                  *ptr1 = input_quant();
                  ptr1++;
                  *ptr1 = input_set();
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);
```

```
                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptrl++;
                *ptrl = input_singleton();
                break;
        case 6:     /*It is alternative  no 6     */
                ptrl++;
                *ptrl = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptrl)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptrl++;
                *ptrl = input_set();
                break;
        case 7:     /*It is alternative  no 7     */
                ptrl++;
                *ptrl = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptrl)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptrl++;
                *ptrl = input_singleton();
                ptrl++;
                *ptrl = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptrl)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
```

```
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_exp7\n");
#endif

      return((int) ptr);

}/* end input_exp7*/



input_aggcall()
{
AGGCALL_TYPE      ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_aggcall\n");
#endif

      ptr = (AGGCALL_TYPE)malloc(sizeof(struct aggcall_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  ptr1++;
                  *ptr1 = input_set();
                  ptr1++;
```

```
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 2:     /*It is alternative  no 2     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_set();
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);
```

```
                    break;
               default:printf("ERROR : Wrong lexeme type
                    generated\n");
          }/* end switch */
          break;
case 3:     /*It is alternative  no 3     */
          ptr1++;
          *ptr1 = (int )malloc(sizeof(OPR_TYPE));
          fscanf(fp, "%d", &temptype);
          ((OPR_TYPE *) *ptr1)->type = temptype;
          switch(temptype)
          {
               case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
               default:printf("ERROR : Wrong lexeme type
                    generated\n");
          }/* end switch */
          ptr1++;
          *ptr1 = (int )malloc(sizeof(OPR_TYPE));
          fscanf(fp, "%d", &temptype);
          ((OPR_TYPE *) *ptr1)->type = temptype;
          switch(temptype)
          {
               case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *ptr1 = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
               default:printf("ERROR : Wrong lexeme type
                    generated\n");
          }/* end switch */
          ptr1++;
          *ptr1 = input_set();
          ptr1++;
          *ptr1 = (int )malloc(sizeof(OPR_TYPE));
          fscanf(fp, "%d", &temptype);
          ((OPR_TYPE *) *ptr1)->type = temptype;
          switch(temptype)
          {
               case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
               default:printf("ERROR : Wrong lexeme type
                    generated\n");
          }/* end switch */
          break;
case 4:     /*It is alternative  no 4     */
          ptr1++;
          *ptr1 = (int )malloc(sizeof(OPR_TYPE));
          fscanf(fp, "%d", &temptype);
          ((OPR_TYPE *) *ptr1)->type = temptype;
          switch(temptype)
```

```
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_singleton();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_mtuple();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
```

```
        break;
case 5:     /*It is alternative  no 5     */
        ptrl++;
        *ptrl = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptrl)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptrl);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptrl++;
        *ptrl = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptrl)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptrl);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptrl++;
        *ptrl = input_singleton();
        ptrl++;
        *ptrl = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptrl)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptrl);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptrl++;
        *ptrl = input_mtuple();
        ptrl++;
        *ptrl = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptrl)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptrl);
                    fscanf(fp, "%s", name);
```

```
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
case 6:     /*It is alternative  no 6    */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = input_singleton();
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
case 7:     /*It is alternative  no 7    */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
```

```
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_singleton();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
      default:printf("ERROR : Wrong lexeme type
            generated\n");
}/* end switch */
ptr1++;
*ptr1 = input_mtuple();
ptr1++;
*ptr1 = (int )malloc(sizeof(OPR_TYPE));
fscanf(fp, "%d", &temptype);
((OPR_TYPE *) *ptr1)->type = temptype;
switch(temptype)
{
      case 1:     /*It is an operator     */
            tptr = (int *) (*ptr1);
            fscanf(fp, "%s", name);
            *tptr = (int )malloc(strlen(name)+1);
            strcpy((char *) (*tptr), name);

            break;
```

```
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                    }/* end switch */
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_aggcall\n");
#endif

      return((int) ptr);

}/* end input_aggcall*/



input_comp()
{
COMP_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_comp\n");
#endif

      ptr = (COMP_TYPE)malloc(sizeof(struct comp_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
```

```
                default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
        case 3:     /*It is alternative  no 3     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        case 4:     /*It is alternative  no 4     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        case 5:     /*It is alternative  no 5     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                        case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                        default:printf("ERROR : Wrong lexeme type
                                generated\n");
                }/* end switch */
                break;
        case 6:     /*It is alternative  no 6     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
```

```
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_comp\n");
#endif

        return((int) ptr);

}/* end input_comp*/



input_quant()
{
QUANT_TYPE  ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_quant\n");
#endif

        ptr = (QUANT_TYPE)malloc(sizeof(struct quant_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
                    switch(temptype)
                    {
                          case 1:     /*It is an operator     */
                                tptr = (int *) (*ptr1);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
              case 2:     /*It is alternative  no 2     */
                    ptr1++;
                    *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((OPR_TYPE *) *ptr1)->type = temptype;
```

```
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 3:     /*It is alternative  no 3     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        break;
case 4:     /*It is alternative  no 4     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
                    *tptr = (int )malloc(strlen(name)+1);
                    strcpy((char *) (*tptr), name);

                    break;
              default:printf("ERROR : Wrong lexeme type
                    generated\n");
        }/* end switch */
        ptr1++;
        *ptr1 = input_leastmost();
        ptr1++;
        *ptr1 = input_singleton();
        break;
case 5:     /*It is alternative  no 5     */
        ptr1++;
        *ptr1 = (int )malloc(sizeof(OPR_TYPE));
        fscanf(fp, "%d", &temptype);
        ((OPR_TYPE *) *ptr1)->type = temptype;
        switch(temptype)
        {
              case 1:     /*It is an operator     */
                    tptr = (int *) (*ptr1);
                    fscanf(fp, "%s", name);
```

```
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = input_singleton();
                break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_quant\n");
#endif

        return((int) ptr);

}/* end input_quant*/



input_leastmost()
{
LEASTMOST_TYPE    ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_leastmost\n");
#endif

        ptr = (LEASTMOST_TYPE)malloc(sizeof(struct leastmost_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
            case 1:     /*It is alternative  no 1     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
            case 2:     /*It is alternative  no 2     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
```

```
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_leastmost\n");
#endif

      return((int) ptr);

}/* end input_leastmost*/



input_bool()
{
BOOL_TYPE   ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_bool\n");
#endif

      ptr = (BOOL_TYPE)malloc(sizeof(struct bool_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            case 2:     /*It is alternative  no 2     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((OPR_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 1:     /*It is an operator     */
```

```
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_bool\n");
#endif

      return((int) ptr);

}/* end input_bool*/



input_funcid()
{
FUNCID_TYPE       ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_funcid\n");
#endif

      ptr = (FUNCID_TYPE)malloc(sizeof(struct funcid_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(IDENTIFIERF_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((IDENTIFIERF_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 6:     /*It is an identifier   */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_funcid\n");
#endif

      return((int) ptr);
```

```
}/* end input_funcid*/


input_typeid()
{
TYPEID_TYPE       ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_typeid\n");
#endif

      ptr = (TYPEID_TYPE)malloc(sizeof(struct typeid_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
      {
            case 1:     /*It is alternative  no 1     */
                  ptr1++;
                  *ptr1 = (int )malloc(sizeof(IDENTIFIER_TYPE));
                  fscanf(fp, "%d", &temptype);
                  ((IDENTIFIER_TYPE *) *ptr1)->type = temptype;
                  switch(temptype)
                  {
                        case 2:     /*It is an identifier   */
                              tptr = (int *) (*ptr1);
                              fscanf(fp, "%s", name);
                              *tptr = (int )malloc(strlen(name)+1);
                              strcpy((char *) (*tptr), name);

                              break;
                        default:printf("ERROR : Wrong lexeme type
                              generated\n");
                  }/* end switch */
                  break;
            default:printf("ERROR: illegal alternative no.\n");
      }/* end switch */

#ifdef      DEBUG
printf("Out input_typeid\n");
#endif

      return((int) ptr);

}/* end input_typeid*/


input_vblid()
{
VBLID_TYPE  ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_vblid\n");
#endif

      ptr = (VBLID_TYPE)malloc(sizeof(struct vblid_type));
      ptr1 = (int *)ptr;
      fscanf(fp, "%d", ptr);
      switch(ptr->type)
```

```
        {
              case 1:     /*It is alternative  no 1     */
                    ptrl++;
                    *ptrl = (int )malloc(sizeof(IDENTIFIERV_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((IDENTIFIERV_TYPE *) *ptrl)->type = temptype;
                    switch(temptype)
                    {
                          case 5:     /*It is an var identifier */
                                tptr = (int *) (*ptrl);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
                    break;
              default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_vblid\n");
#endif

        return((int) ptr);

}/* end input_vblid*/



input_svfuncall()
{
SVFUNCALL_TYPE    ptr;
int *ptrl;

#ifdef      DEBUG
printf("In input_svfuncall\n");
#endif

        ptr = (SVFUNCALL_TYPE)malloc(sizeof(struct svfuncall_type));
        ptrl = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
              case 1:     /*It is alternative  no 1     */
                    ptrl++;
                    *ptrl = (int )malloc(sizeof(IDENTIFIERF_TYPE));
                    fscanf(fp, "%d", &temptype);
                    ((IDENTIFIERF_TYPE *) *ptrl)->type = temptype;
                    switch(temptype)
                    {
                          case 6:     /*It is an identifier   */
                                tptr = (int *) (*ptrl);
                                fscanf(fp, "%s", name);
                                *tptr = (int )malloc(strlen(name)+1);
                                strcpy((char *) (*tptr), name);

                                break;
                          default:printf("ERROR : Wrong lexeme type
                                generated\n");
                    }/* end switch */
```

```
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = input_singleton();
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      case 2:     /*It is alternative  no 2     */
            ptrl++;
            *ptrl = (int )malloc(sizeof(IDENTIFIERF_TYPE));
            fscanf(fp, "%d", &temptype);
            ((IDENTIFIERF_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 6:     /*It is an identifier   */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptrl++;
            *ptrl = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptrl)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptrl);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);
```

```
                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                ptr1++;
                *ptr1 = input_singleton();
                ptr1++;
                *ptr1 = input_singlist();
                ptr1++;
                *ptr1 = (int )malloc(sizeof(OPR_TYPE));
                fscanf(fp, "%d", &temptype);
                ((OPR_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                    case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                    default:printf("ERROR : Wrong lexeme type
                        generated\n");
                }/* end switch */
                break;
            default:printf("ERROR: illegal alternative no.\n");
        }/* end switch */

#ifdef      DEBUG
printf("Out input_svfuncall\n");
#endif

        return((int) ptr);

}/* end input_svfuncall*/



input_mvfuncall()
{
MVFUNCALL_TYPE    ptr;
int *ptr1;

#ifdef      DEBUG
printf("In input_mvfuncall\n");
#endif

        ptr = (MVFUNCALL_TYPE)malloc(sizeof(struct mvfuncall_type));
        ptr1 = (int *)ptr;
        fscanf(fp, "%d", ptr);
        switch(ptr->type)
        {
            case 1:     /*It is alternative  no 1     */
                ptr1++;
                *ptr1 = (int )malloc(sizeof(IDENTIFIERF_TYPE));
                fscanf(fp, "%d", &temptype);
                ((IDENTIFIERF_TYPE *) *ptr1)->type = temptype;
                switch(temptype)
                {
                    case 6:     /*It is an identifier   */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
```

```
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            ptr1++;
            *ptr1 = input_mtuple();
            ptr1++;
            *ptr1 = (int )malloc(sizeof(OPR_TYPE));
            fscanf(fp, "%d", &temptype);
            ((OPR_TYPE *) *ptr1)->type = temptype;
            switch(temptype)
            {
                  case 1:     /*It is an operator     */
                        tptr = (int *) (*ptr1);
                        fscanf(fp, "%s", name);
                        *tptr = (int )malloc(strlen(name)+1);
                        strcpy((char *) (*tptr), name);

                        break;
                  default:printf("ERROR : Wrong lexeme type
                        generated\n");
            }/* end switch */
            break;
      default:printf("ERROR: illegal alternative no.\n");
   }/* end switch */

#ifdef      DEBUG
printf("Out input_mvfuncall\n");
#endif

      return((int) ptr);

}/* end input_mvfuncall*/
```

```
/*          FILE: main.c            */

/* This is the main function for the automatic generation */

/* Includes for this file     */

#include          <stdio.h>
#include          <sys/types.h>
#include          <sys/dir.h>
#include          <sys/stat.h>
#include          "tabledefs.h"
#include          "tabledecls.h"
#include          "catalog.h"




main(argc, argv)
int argc;
char *argv[];
{

      /* check specified database exists */
      if (argc < VALIDARGS)
      {
            printf("FQLFE:Invoke using an existing database\n");
            exit(0);
      }
      strcpy(dbname, argv[1]);

      /* create names of files for this database     */
      sprintf(dbname_rel, "%s%s", dbname, rel);
      sprintf(dbname_func, "%s%s", dbname, func);
      sprintf(dbname_defs, "%s%s", dbname, defs);

      /* change to the local .FQLFE directory  */
      check_FQLFE(dbname_defs);

      /* create the table of catalogue information   */
      create_catalog();

      /* use this to create definitions of functions       */
      create_defsfile();
      exit(0);
}



/*This forks a child process which invokes the ingres database
with the given name. It takes its commands from infile and writes
its output to outfile*/

dbexec(infile, outfile)
char *infile, *outfile;
{
      int status, fd, fd2, pid;

      switch (pid = fork() )
        {
            case FAIL:
                        system_error("FORK failed\n");
            case CHILD:
```

```
                        /*redirect input */
                        fd = open(infile, 0);
                        close(0);
                        dup(fd);
                        close(fd);

                        /* redirect output */
                        fd2 = open(outfile, 1);
                        close(1);
                        dup(fd2);
                        close(fd2);

                        /* call the Ingres DBMS       */
                        execlp("ingres", "ingres", dbname,
                              (char *) 0);
                        system_error("EXECLP failed\n");

                PARENT:
                        wait(&status);
        }
}


/*This routine uses the result file and extracts the relevant
information about the relations and the attributes.*/

skiplines(n)
int n;
{
      int i;

        for (i = 0; i < n; i++)
        {
                fgets(ignore, MAXLINE, fp);
        }
}


/*This creates a table about the database from the system
catalog*/

create_catalog()
{
      /* local variables      */
      int i, k, n, j;
      char name1[25], name2[25], name3[25];
      nullstring = "";

      /* open file to contain catalogue requests      */
      if ((fp = fopen ("catalog_request", "w")) == NULL)
            FQLFE_error("Unable to open file catalog_request");
      /* put out relevant requests */
      sprintf(retrieve2, "help\n");
      fputs(retrieve2, fp);
      fputs(gostate, fp);
      fclose(fp);

      /* call database execution routine */
      dbexec("catalog_request", "result");
```

```
/*open result file */
if ((fp = fopen ("result", "r")) == NULL)
      FQLFE_error("Unable to open file result");

/* now scan out relation names for this database*/
skiplines(12);
k = 0;
fscanf(fp, "%s%s%s ", name1, name2, name3);
while (strcmp(name1, "continue") != 0)
{
      if (strcmp(name3, "table") == 0)
      {
            strcpy(catalog.relation[k].relname, name1);
            k++;
      }
      fscanf(fp, "%s%s%s ", name1, name2, name3);
}
relcount = k;
fclose(fp);

/*now scan out attribute names for this relation */
for(i = 0; i < relcount; i++)
{
      /* open file to contain catalogue requests     */
      if ((fp = fopen("catalog_request", "w")) == NULL)
            FQLFE_error("Unable to open file
                  catalog_request");

      /* write out requests to file      */
      sprintf(retrieve1, "help %s\n",
            catalog.relation[i].relname);
      fputs(retrieve1, fp);
      fputs(gostate, fp);
      fclose(fp);

      /* call database execution routine */
      dbexec("catalog_request", "result");

      /*open result file */
      if ((fp = fopen("result", "r")) == NULL)
                  FQLFE_error("Unable to open file result");

      /* extract relevant information    */
      skiplines(24);
      fscanf(fp, "%s", name1);
      fscanf(fp, "%s", name2);
      fscanf(fp, "%s", ignore);
      fscanf(fp, "%d", ignore);
      j = 0;
      while (strcmp(name1, "Secondary") != 0)
      {
            strcpy(catalog.relation[i].attribute[j].attrname,
                        name1);
            if (strcmp(name2, "integer") == 0)
                  strcpy(name2, "INT");
            else
                  strcpy(name2, "STR");
            strcpy(catalog.relation[i].attribute[j].attrtype,
                        name2);
            j++;
            fscanf(fp, "%s", name1);
            fscanf(fp, "%s", name2);
            fscanf(fp, "%s", ignore);
```

```
                fscanf(fp, "%d", ignore);
            }/*end of attribute loop */
            fclose(fp);

            /* store the count of number of attributes for
                relation    */
            catalog.relation[i].attrcount = j;

        }/*end of relation loop */
}


/*This routine checks to see if the database dbname already has
descriptions for it in the directory $HOME/FAP*/

check_FQLFE(name)
char name[];
{

        /* local variables      */
        int fd;
        struct stat stbuf;
        char wd[100], dir[100];

        /* check for environment variable HOME    */
        if (( h = getenv("HOME")) == ((char *) NULL))
        {
            FQLFE_error("No home directory specified\n");
            exit(0);
        }

        /* generate directory name    */
        sprintf(dir, "%s/%s", h, ".FQLFE");

        /* change to the directory $HOME/FAP      */
        if (chdir(dir) == FAIL)
        {

            /*assume it does not exist and create new one  */
            sprintf(syscomm, "mkdir %s", dir);
            system(syscomm);

            /* attempt to cahnge to the created directory  */
            if (chdir(dir) == FAIL)
            {
                FQLFE_error("Unable to change to directory
                            $HOME/FAP\n");
                exit(0);
            }
          }


        /*check file exists     */
        if (stat(name, &stbuf) == FAIL)
        {
                return(1);    /* ie file not there */
        }
          else
        {
                return(0);    /* ie file is there */
        }
}
```

```
/*This routine creates the file of definitions*/

create_defsfile()
{
      /* local variables      */
      int j;
      FILE *fp;

      /* open file to contain generated function definitions     */
      if ((fp = fopen(dbname_defs, "w")) == NULL)
            FQLFE_error("Unable to create definitions file\n");

      /* process catalogue information table and extract details
      */
      for (i = 0; i < relcount; i++)
      {

            /* output base function declaration statement  */
            fprintf(fp, "DECLARE %s( )-> ENTITY.\n",
                        catalog.relation[i].relname);
            for (j = 0; j < catalog.relation[i].attrcount; j++)
            {

                  /* output nonbase function declaration stmnt   */
                  fprintf(fp, "DECLARE %s(%s)-> %s.\n",
                        catalog.relation[i].attribute[j].attrname,
                        catalog.relation[i].relname,
                        catalog.relation[i].attribute[j].attrtype);
            }
      }
      fputs(gostate, fp);
      fputs(quitstate, fp);
      fclose(fp);

      /* Call the FQLFE system to process the definitions file   */
      sprintf(syscomm, "fqlfe %s <%s >temp.out", dbname,
            dbname_defs);

      system(syscomm);

}/* end of create_defsfile*/



/*This is an error procesing routine*/

FQLFE_error(mess)
char * mess;
{
        printf("FQLFE: Error during automatic function definition
                  phase \n");
        printf("%s\n", mess);
      exit(1);
}



/*This is an error procesing routine for system errors*/

system_error(mess)
```

```
char * mess;
{
        printf("FQLFE AUTOGEN: SYSTEM ERROR\n");
        printf("%s\n", mess);
      exit(1);
}
```

APPENDIX E: RESULTS OF EXAMPLE QUERIES

EXAMPLE DATABASE

empno	name	salary	manage	deptno
1	SHARPE	28000	1	1
2	JONES	25000	1	3
3	ANDERSON	10000	2	2
4	SMITH	12000	2	3
5	HARRIS	13000	4	4
6	WHITE	14000	4	4
7	BROWN	12000	3	2
8	BLACK	26000	7	2
9	WILLOW	10000	4	4
10	DIXON	8000	4	4
11	FOX	21000	4	4
12	BAKER	11000	4	4
13	FLOWERS	9000	4	4
14	STEEL	11000	4	4
15	TOWERS	12000	4	4
16	GARDNER	13000	4	4
17	BELL	18000	4	4
18	BENSON	18000	18	5
19	HART	19000	8	5
20	PETERS	24000	8	6

deptno	itemno	vol
1	11	700
2	2	600
2	10	50
3	2	50
4	5	200
5	6	25
5	7	50
5	8	600
6	1	25

compno	deptno	itemno	vol
1	5	6	100
1	6	1	10
2	5	7	250
2	2	2	1000
2	2	8	1000
2	6	1	500
2	2	9	100
2	2	10	700
3	5	7	100
3	5	8	1000
4	1	11	1000
5	3	4	100
5	4	5	750
6	3	3	200

compno	name	address
1	SMITH & CO	LONDON
2	JONES LTD	MANCHESTER
3	HALL & SONS	BIRMINGHAM
4	WILLIAMS & CO	LONDON
5	JOHNSON & CO	LONDON
6	SINGER LTD	BIRMINGHAM

deptno	name	floor
1	TOY	1
2	BOOK	2
3	RECORD	2
4	SHOE	3
5	STATIONERY	4
6	PHOTOGRAPHY	5

itemno	name	type
1	CAMERA	A
2	PAPERBACK	B
3	SINGLE	C
4	DRESSES	B
5	SHOES	A
6	PEN	A
7	PENCIL	B
8	ERASER	A
9	STAPLES	D
10	PAPER	A
11	TOYS	A

RESULTS OF EXAMPLE QUERIES

```
FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee SUCH THAT
**     name(dept(employee)) = "TOY"
**PRINT name(employee)
**\go

Transformed query is.......

range of d is department
range of e is employee
retrieve  ( e.name  )  where
```

```
      e.deptno  = d.deptno  and d.name  = "TOY"

executing transformed query......

result......

|name                      |
|--------------------------|
|SHARPE                    |
|--------------------------|


**\quit

.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH item SUCH THAT
**    FOR SOME sold(item)
**          floor(department) = 2
**PRINT name(item), name(department)

**\go

Transformed query is.......


range of d is department
range of i is item
range of s is sales
retrieve  ( i.name, d.name  )
where d.floor  =  2  and s.itemno  = i.itemno  and
      d.deptno  = s.deptno

executing transformed query......

result......

|name                      |name                      |
|--------------------------|--------------------------|
|PAPERBACK                 |BOOK                      |
|PAPER                     |BOOK                      |
|PAPERBACK                 |RECORD                    |
|--------------------------|--------------------------|


**\quit
```

```
.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH e IN employee SUCH THAT
**    FOR  SOME e1 in employee SUCH THAT
**          name(e1) = "ANDERSON" AND
**          managerno(e1) = empno(e)
**PRINT salary(e)
**\go

Transformed query is.......


range of e is employee
range of e1 is employee
retrieve (e.salary)
where e1.name = "ANDERSON" and
      e1.managerno = e.empno

executing transformed query......

result......


|salary   |
|---------|
|  25000  |
|---------|


**\quit

.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************
```

```
ready
**FOR EACH e IN employee SUCH THAT
**    FOR SOME e1 in employee
**          salary(e) > salary(e1) AND
**          managerno(e) = empno(e1)
**PRINT name(e)
**\go

Transformed query is.......


range of e is employee
range of e1 is employee
retrieve (e.name) where
      e.managerno = e1.empno and
      e.salary > e1.salary


executing transformed query......

result......


|name                       |
|---------------------------|
|BROWN                      |
|GARDNER                    |
|HARRIS                     |
|WHITE                      |
|BELL                       |
|FOX                        |
|BLACK                      |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH e in employee SUCH THAT
**    FOR SOME e1 in employee
**          managerno(e) = empno(e1) AND
**          salary(e) = salary(e1)
**PRINT name(e)
**\go

Transformed query is.......
```

```
range of e is employee
range of e1 is employee
retrieve (e.name) where
      e.managerno = e1.empno and
      e.salary = e1.salary

executing transformed query......

result......


|name                       |
|---------------------------|
|TOWERS                     |
|BENSON                     |
|SHARPE                     |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH e in employee SUCH THAT
**    FOR SOME e1 in employee
**          managerno(e1) = empno(e)   AND
**          deptno(e) = deptno(e1)
**PRINT name(e)
**\go

Transformed query is.......


range of e is employee
range of e1 is employee
retrieve (e.name) where
      e.deptno = e1.deptno AND
      e.empno = e1.managerno

executing transformed query......

result......


|name                       |
|---------------------------|
|SHARPE                     |
```

```
|JONES                      |
|ANDERSON                   |
|BROWN                      |
|BENSON                     |
|---------------------------|

**\quit

.....EXIT FQLFE.
```

```
FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************
```

```
ready
**FOR EACH employee SUCH THAT
**     FOR SOME dept(employee)
**           AVERAGE (salary(employee)
**                 OVER deptno(employee))
**                       > 25000
**PRINT name(department)
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (d.name) where
      d.deptno = e.deptno AND
      avg(e.salary by e.deptno) > 25000

executing transformed query......

result......


|name                       |
|---------------------------|
|TOY                        |
|---------------------------|


**\quit

.....EXIT FQLFE.
```

```
FQLFE.....

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH e in employee SUCH THAT
**     FOR SOME e1 in employee
**           managerno(e1) = manager(e) AND
**           count(manager(e1) OVER e1) > 10
**PRINT name(e), salary(e)
**\go

Transformed query is.......


range of e is employee
range of e1 is employee
retrieve (e.name, e.salary) where
      e.empno = e1.managerno AND
      count(e1.empno by e1.managerno) > 10


executing transformed query......

result......


|name                          |salary|
|------------------------------|
|SMITH                         | 12000|
|------------------------------|


**\quit

.....EXIT FQLFE.

FQLFE.....

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH e IN employee SUCH THAT
**     FOR SOME e1 in employee
**           managerno(e) = empno(e1) AND
**           name(department) = "SHOE" AND
**           deptno(e) = deptno(department) AND
**           salary(e) > 13000
**PRINT name(e), name(e1)
**\go
```

```
Transformed query is.......

range of e is employee
range of el is employee
range of d is department
retrieve (e.name, el.name) where
      e.managerno = el.empno AND
      e.deptno = d.deptno AND
      d.name = "SHOE" AND
      e.salary > 13000

executing transformed query......

result......

|name                          |name                          |
|------------------------------------------------------------|
|BELL                          |SMITH                         |
|FOX                           |SMITH                         |
|WHITE                         |SMITH                         |
|------------------------------------------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH  employee SUCH THAT
**    salary(employee) >
**    MAXIMUM(salary(employee) SUCH THAT
**          name(dept(employee)) = "SHOE")
**PRINT name(employee)
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (e.name) where
      e.salary > max(e.salary where e.deptno = d.deptno
      AND d.name = "SHOE")


executing transformed query......

result......
```

```
|name                       |
|---------------------------|
|SHARPE                     |
|JONES                      |
|BLACK                      |
|PETERS                     |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH department SUCH THAT
**     FOR SOME employee
**           TOTAL (salary(employee)
**                 OVER deptno(sales) SUCH THAT
**                       deptsells(employee))
**           > 40000
**           AND name(itemsold(department)) = "PAPERBACK"
**PRINT name(department)
**\go

Transformed query is.......


range of s is sales
range of e is employee
range of d is department
range of i is item
retrieve (d.name) where
       sum(e.salary by s.deptno where
             s.deptno = e.deptno)
       > 40000 and
       i.name = "PAPERBACK" and
       i.itemno = s.itemno and
       s.deptno = d.deptno

executing transformed query......

result......


|name                       |
|---------------------------|
```

```
|BOOK                        |
|----------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

********************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH item SUCH THAT
**     FOR SOME supplies(item)
**           name(item) = "PEN"
**PRINT name(supplier).
**\go

Transformed query is.......


range of s is supply
range of s1 is supplier
range of i is item
retrieve (s1.name) where
       i.name = "PEN" and
       i.itemno = s.itemno and
       s.compno = s1.compno


executing transformed query......

result......


|name                        |
|----------------------------|
|SMITH & CO                  |
|----------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......
```

```
Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH item SUCH THAT
**     FOR SOME supplies(item)
**           name(item) NE "PEN"
**PRINT name(supplier).
**\go

Transformed query is.......

range of s is supply
range of s1 is supplier
range of i is item
retrieve (s1.name) where
       i.name != "PEN" and
       i.itemno = s.itemno and
       s.compno = s1.compno

executing transformed query......

result......


|name                       |
|---------------------------|
|SMITH & CO                 |
|JONES LTD                  |
|HALL & SONS                |
|JOHNSON & CO               |
|SINGER LTD                 |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH item SUCH THAT
**     FOR SOME supplied(item)
**           COUNT (compno(supply)
**                 OVER itemno(supply)) = 1
```

```
**PRINT name(item).
**\go

Transformed query is.......

range of s is supply
range of i is item
retrieve (i.name) where
      i.itemno = s.itemno and
      count(s.compno by s.itemno)   = 1


executing transformed query......

result......


|name                       |
|---------------------------|
|DRESSES                    |
|PAPERBACK                  |
|PEN                        |
|SHOES                      |
|SINGLE                     |
|STAPLES                    |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH supply SUCH THAT
**    FOR SOME comp(supply)
**          COUNT (compno(supply)
**                BY itemno(supply)) = 1
**PRINT name(supplier).

**\go

Transformed query is.......


range of s is supply
range of s1 is supplier
retrieve (s1.name) where
      s1.compno = s.compno and
```

```
      count(s.compno by s.itemno) = 1

executing transformed query......

result......

|name                       |
|---------------------------|
|JOHNSON & CO               |
|JONES LTD                  |
|SINGER LTD                 |
|SMITH & CO                 |
|WILLIAMS & CO              |
|---------------------------|

**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH supply SUCH THAT
**    FOR SOME comp(supply)
**          COUNT (itemno(supply) OVER compno(supply)
**          SUCH THAT
**                (COUNT (compno(supply)
**                      OVER itemno(supply))
**                = 1)
**          >= 3
**PRINT name(supplier).
**\go

Transformed query is.......


range of s is supply
range of s1 is supplier
range of i is item
retrieve (s1.name) where
      s1.compno = s.compno and
      count(s.itemno by s.compno where
            count(s.compno by s.itemno) = 1 )
      >=3


executing transformed query......

result......
```

```
|name                       |
|---------------------------|
|JONES LTD                  |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH item SUCH THAT
**     sold(item)
**PRINT name(item), type(item), name(department),
**     floor(department).

**\go

Transformed query is.......

range of i is item
range of s is sales
range of d is department
retrieve (i.name, i.type, d.name, d.floor) where
       i.itemno = s.itemno and
       s.deptno = d.deptno


executing transformed query......

result......

|name                       |type|name                       |floor |
|---------------------------|----|---------------------------|------|
|PAPERBACK                  |B   |BOOK                       |     2|
|PAPERBACK                  |B   |RECORD                     |     2|
|SHOES                      |A   |SHOE                       |     3|
|ERASER                     |A   |STATIONERY                 |     4|
|PEN                        |A   |STATIONERY                 |     4|
|PENCIL                     |B   |STATIONERY                 |     4|
|CAMERA                     |A   |PHOTOGRAPHY                |     5|
|---------------------------|----|---------------------------|------|
|
```

```
**\quit

.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH employee
**PRINT AVERAGE (salary(employee)
**     SUCH THAT name(dept(employee)) = "SHOE").
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (a = avg(e.salary where
       d.name = "SHOE" and
       d.deptno = e.deptno))

executing transformed query......

result......


|a          |
|-----------|
| 12727.273|
|-----------|


**\quit

.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************
```

```
ready
**FOR EACH employee SUCH THAT
**     FOR SOME dept(employee)
**PRINT name(department),
**     AVERAGE(salary(employee)
**          OVER deptno(employee) SUCH THAT
**               dept(employee)).
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (d.name, a = avg(e.salary by e.deptno)) where
      e.deptno = d.deptno
executing transformed query......

result......


|name                      | a         |
|--------------------------------------|
|BOOK                      | 16000.000|
|PHOTOGRAPHY               | 24000.000|
|RECORD                    | 18500.000|
|SHOE                      | 12727.273|
|STATIONERY                | 18500.000|
|TOY                       | 28000.000|
|--------------------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee SUCH THAT
**     dept(employee)
**PRINT name(department), floor(department),
**     AVERAGE (salary(employee) OVER deptno(employee))
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (d.name, d.floor,
```

```
      a = avg(e.salary by e.deptno))
      where e.deptno = d.deptno


executing transformed query......

result......

|name                          |floor     | a     |
|------------------------------------------|----------|
|BOOK                          |         2| 16000.000|
|PHOTOGRAPHY                   |         5| 24000.000|
|RECORD                        |         2| 18000.000|
|SHOE                          |         3| 12727.273|
|STATIONERY                    |         4| 18500.000|
|TOY                           |         1| 28000.000|
|------------------------------------------|----------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH supply SUCH THAT
**     FOR SOME item SUCH THAT
**           FOR SOME comp(supply)
**                 TOTAL (vol(supply)
**                       OVER compno(supply)
**                             SUCH THAT supplied(item)
**                                   AND type(item) = "A")
**                 +
**                 TOTAL (vol(supply)
**                       OVER compno(supply)
**                             SUCH THAT supplied(item)
**                                   AND type(item) = "B")
**                 > 1000
**PRINT name(supplier).
**\go

Transformed query is.......

range of s is supply
range of i is item
range of s1 is supplier
retrieve (s1.name) where
      s1.compno = s.compno and
```

```
      sum(s.vol by s.compno where s.itemno = i.itemno
            and i.type = "A")
      +
      sum(s.vol by s.compno where s.itemno = i.itemno
            and i.type = "B")
      > 1000

executing transformed query......

result......

|name                       |
|---------------------------|
|HALL & SONS                |
|JONES LTD                  |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee SUCH THAT
**    FOR SOME dept(employee)
**          name(department) = "SHOE"
**PRINT name(employee,
**    salary(employee) - AVERAGE (salary(employee)
**          OVER deptno(employee) SUCH THAT
**                name(dept(employee)) = "SHOE").
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (e.name,
      a = e.salary - avg(e.salary by e.deptno where
            e.deptno = d.deptno and
                  d.name = "SHOE")) where
      e.deptno = d.deptno
      and d.name = "SHOE"

executing transformed query......

result......
```

```
|name                          |a          |
|------------------------------------------|
|BAKER                         | -1727.273|
|BELL                          |  5272.727|
|DIXON                         | -4727.273|
|FLOWERS                       | -3727.273|
|FOX                           |  8272.727|
|GARDNER                       |   272.727|
|HARRIS                        |   272.727|
|STEEL                         | -1727.273|
|TOWERS                        |  -727.273|
|WHITE                         |  1272.727|
|WILLOW                        | -2727.273|
|------------------------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee SUCH THAT
**    FOR SOME dept(employee) SUCH THAT
**          name(department) = "SHOE"
**PRINT name(employee),
**    salary(employee) - AVERAGE (salary(employee))
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (e.name, a = e.salary - avg(e.salary)) where
      d.name = "SHOE" and d.deptno = e.deptno

executing transformed query......

result......


|name                          |a          |
|------------------------------------------|
|BAKER                         | -4700.000|
|BELL                          |  2300.000|
|DIXON                         | -7700.000|
|FLOWERS                       | -6700.000|
```

```
|FOX                          |  5300.000|
|GARDNER                      | -2700.000|
|HARRIS                       | -2700.000|
|STEEL                        | -4700.000|
|TOWERS                       | -3700.000|
|WHITE                        | -1700.000|
|WILLOW                       | -5700.000|
|-------------------------------------|

**\quit

.....EXIT FQLFE.



FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee
**PRINT name(employee),
**     salary(employee) - AVERAGE (salary(employee)
**          OVER deptno(employee))
**\go

Transformed query is.......


range of e is employee
retrieve (e.name,
      a = e.salary - avg(e.salary by e.deptno))


executing transformed query......

result......


|name                         |a         |
|-------------------------------------|
|ANDERSON                     | -6000.000|
|BAKER                        | -1727.273|
|BELL                         |  5272.727|
|BENSON                       |  -500.000|
|BLACK                        | 10000.000|
|BROWN                        | -4000.000|
|DIXON                        | -4727.273|
|FLOWERS                      | -3727.273|
|FOX                          |  8272.727|
|GARDNER                      |   272.727|
|HARRIS                       |   272.727|
|HART                         |   500.000|
|JONES                        |  6500.000|
```

```
|PETERS                          |      0.000|
|SHARPE                          |      0.000|
|SMITH                           |  -6500.000|
|STEEL                           |  -1727.273|
|TOWERS                          |   -727.273|
|WHITE                           |   1272.727|
|WILLOW                          |  -2727.273|
|-----------------------------------------|

**\quit

.....EXIT FQLFE.

FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

*******************WELCOME TO THE FQLFE SYSTEM***************


ready
**FOR EACH supply SUCH THAT
**     comp(supply)
**PRINT name(supplier),
**     AVERAGE (COUNT (itemno(supply)
**           OVER deptno(supply),
**                 compno(supply))
**     OVER compno(supply))
**\go

Transformed query is.......

range of s is supply
range of s1 is supplier
retrieve (s1.name,
       a = avg(count(s1.itemno by s.deptno, s.compno)
             by s.compno)) where
       s1.compno = s.compno

executing transformed query......

result......


|name                            |a          |
|-----------------------------------------|
|HALL & SONS                     |   2.000|
|JOHNSON & CO                    |   1.000|
|JONES LTD                       |   2.500|
|SINGER LTD                      |   1.000|
|SMITH & CO                      |   1.000|
|WILLIAMS & CO                   |   1.000|
|-----------------------------------------|


**\quit

.....EXIT FQLFE.
```

```
FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH employee SUCH THAT
**     dept(employee)
**PRINT name(department), AVERAGE (salary(employee)
**                                 OVER deptno(employee) SUCH THAT
**                                       salary(employee) >
**                                       AVERAGE (salary(employee)
**                                             OVER deptno(employee)))
**\go

Transformed query is.......


range of e is employee
range of d is department
retrieve (d.name, a = avg(e.salary by e.deptno where
       e.salary > avg(e.salary by e.deptno)))
       and d.deptno = e.deptno


executing transformed query......

result......


|name                          |a          |
|------------------------------------------|
|BOOK                          | 26000.000|
|PHOTOGRAPHY                   |     0.000|
|RECORD                        | 25000.000|
|SHOE                          | 16500.000|
|STATIONERY                    | 19000.000|
|TOY                           |     0.000|
|------------------------------------------|
**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...
```

```
**********WELCOME TO THE FQLFE SYSTEM**********


ready
**FOR EACH employee
**PRINT AVERAGE (AVERAGE (salary(employee)
**     OVER deptno(employee)))
**\go

Transformed query is.......

range of e is employee
retrieve (a = avg(avg(e.salary by e.deptno)))

executing transformed query......

result......


|a           |
|------------|
| 19621.212  |
|------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

**********WELCOME TO THE FQLFE SYSTEM**********


ready
**FOR EACH employee
**PRINT name(employee),
**     AVERAGE (salary(employee) OVER deptno(employee)),
**          salary(employee) - AVERAGE (salary(employee)
**                                      OVER deptno(employee))
**
**\go

Transformed query is.......


range of e is employee
retrieve (e.name,
      a = avg(e.salary by e.deptno),
      b = e.salary - avg(e.salary by e.deptno))


executing transformed query......
```

```
result......

|name                      |a          |b          |
|--------------------------|-----------|-----------|
|ANDERSON                  | 16000.000| -6000.000|
|BAKER                     | 12727.273| -1727.273|
|BELL                      | 12727.273|  5272.727|
|BENSON                    | 18500.000|  -500.000|
|BLACK                     | 16000.000| 10000.000|
|BROWN                     | 16000.000| -4000.000|
|DIXON                     | 12727.273| -4727.273|
|FLOWERS                   | 12727.273| -3727.273|
|FOX                       | 12727.273|  8272.727|
|GARDNER                   | 12727.273|   272.727|
|HARRIS                    | 12727.273|   272.727|
|HART                      | 18500.000|   500.000|
|JONES                     | 18500.000|  6500.000|
|PETERS                    | 24000.000|     0.000|
|SHARPE                    | 28000.000|     0.000|
|SMITH                     | 18500.000| -6500.000|
|STEEL                     | 12727.273| -1727.273|
|TOWERS                    | 12727.273|  -727.273|
|WHITE                     | 12727.273|  1272.727|
|WILLOW                    | 12727.273| -2727.273|
|--------------------------|-----------|-----------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH supply SUCH THAT
**     FOR SOME supplier
**           COUNT(compno(supply) OVER compno(supplier)
**                 SUCH THAT name(suppitem(supply)) = "PEN")
**           = 0
**PRINT name(supplier)
**\go

Transformed query is.......

range of i is item
range of s is supply
range of s1 is supplier
retrieve (s1.name) where
      count(s.compno by s1.compno where
```

```
            s1.compno = s.compno and
            s.itemno = i.itemno and
            i.name = "PEN" ) = 0

executing transformed query......

result......

|name                       |
|---------------------------|
|HALL & SONS                |
|JOHNSON & CO               |
|JONES LTD                  |
|SINGER LTD                 |
|WILLIAMS & CO              |
|---------------------------|


**\quit

.....EXIT FQLFE.




FQLFE......

Loading up existing relational description of database...

Loading up existing functional description of database...

******************WELCOME TO THE FQLFE SYSTEM***************



ready
**FOR EACH item SUCH THAT
**     FOR SOME sold(item)
**           COUNT (deptno(sales) OVER itemno(sales)
**                 SUCH THAT floor(floor(department)
**                 = 2) = 0
**PRINT name(item)
**\go

Transformed query is.......

range of d is department
range of s is sales
range of i is item
retrieve (i.name) where
      d.deptno = s.deptno and
      i.itemno = s.itemno and
      count(s.deptno by s.itemno where
      s .deptno = d.deptno and
      d.floor = 2) = 0


executing transformed query......

result......
```

```
|name                       |
|---------------------------|
|CAMERA                     |
|ERASER                     |
|PEN                        |
|PENCIL                     |
|SHOES                      |
|---------------------------|


**\quit

.....EXIT FQLFE.
```