

Automated Deduction

M.Toygar Karadeniz

Sample Input File

```
# all x P(x) or exists y (Q(y) and Z(y))
# all x(P(x,y)) or all z(exists y (Q(y,z) and Z(y,a) or X))
# all x(P(x,y)) and all x(exists y (Q(y,x) and Z(x,a) or X))
# all x P(x) or not (exists y (Q(y) and Z(y)))
# all x (P(x) and R(x)) or exists y (Q(y) and Z(y))
```

```
all x P(x,a) or exists y (Q(y) and not Q(x))
not P(x,a)
```

Sample Output File

```
-----
OS Time: 15:27:21
OS Date: 12/30/99
```

```
all
```

```
*****0'th Parse
id(x)id(P)(id(x),id(a)) or exists id(y)(id(Q)(id(y)) and not id(Q)(id(x))) not
```

```
*****1'th Parse
id(P)(id(x),id(a))
***** Parse Ok, Now Tree Check
```

```
*****0'th Tree Traversal - Normal Traversal
or
id(x) all
id(P) predicate (a)
id(x)
id(a) fullstop
id(y) exists
and
id(Q) predicate (y)
id(y) fullstop
not
id(Q) predicate (x)
id(x) fullstop
*****
```

```
*****1'th Tree Traversal - Normal Traversal
not
id(P) predicate (a)
id(x)
id(a) fullstop
*****
```

```
*****0'th Tree Traversal - After Prenex
id(x) all
id(y) exists
or
id(P) predicate (a)
id(x)
id(a) fullstop
and
id(Q) predicate (y)
id(y) fullstop
id(Q) predicate (x) not
id(x) fullstop
*****
```

```
*****1'th Tree Traversal - After Prenex
id(P) predicate (a) not
id(x)
```

id(a) fullstop

****0'th Tree Traversal - After Conjunctive
id(x) all

id(y) exists
and
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (y)
id(y) fullstop
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (x) not
id(x) fullstop

****1'th Tree Traversal - After Conjunctive
id(P) predicate (a) not
id(x)
id(a) fullstop

****0'th Tree Traversal - After Skolem
and
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (f)
id(f) function (x)
id(x) fullstop
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (x) not
id(x) fullstop

****1'th Tree Traversal - After Skolem
id(P) predicate (a) not
id(x)
id(a) fullstop

****0'th Tree Traversal - After Minimize
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (f)
id(f) function (x)
id(x) fullstop

****1'th Tree Traversal - After Minimize
id(P) predicate (a) not
id(x)
id(a) fullstop

****2'th Tree Traversal - After Minimize
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(Q) predicate (x) not
id(x) fullstop

```
****Tree 3 After Resolution 0 with 1
id(Q) predicate (f)
id(f) function (x)
id(x) fullstop
*****
```

```
****Tree 4 After Resolution 0 with 2
```

```
or
id(P) predicate (a)
id(x)
id(a) fullstop
id(P) predicate (a)
id(x)
id(a) fullstop
*****
```

```
****Tree 5 After Resolution 1 with 2
id(Q) predicate (x) not
id(x) fullstop
*****
```

```
****Tree 6 After Resolution 0 with 5
id(P) predicate (a)
id(x)
id(a) fullstop
*****
```

```
****Contradiction Detected in 1 with 6
```

```
****Resolution Nok - Unsatisfiable
```

Source Codes

```
// lexer.h
#define VALID 0
#define INVALID 1
#define TREES 50

#define NONE 0
#define QUANTIFIERS 1
#define RENAME 2
#define _IMPLIES 3
#define _NOT 4
#define REMOVENOT 5
#define REPLACECONST 6
#define REPLACEFUNC 7
#define FINDOR 8
#define TREECOPY 9
#define PREDICATES 10
#define FREE 11

#define NOTHING 99
#define ID 100
#define ALL 101
#define EXISTS 102
#define OR 103
#define AND 104
#define NOT 105
#define IMPLIES 106
#define LEFTP 107
#define RIGHTP 108
#define COMMA 109
#define PREDICATE 110
#define FUNCTION 111
#define CONSTANT 112
#define EOFFILE 999

typedef int token;
enum direction { left, right, both };
typedef struct node
```

```

{
    token tkn;
    token qual;
    bool fullstop;
    bool not;
    node *left;
    node *right;
    node *parent;
    node *end;
    char id[32];
} mystruct;
typedef struct substitution
{
    bool dummy;
    node *stable;
    node *variable;
    substitution *next;
} mysubstitution;
int clause(node **nd);
node *createnode(token tkn, char *id);
int parse(char *filename,node **nd);
token lexer(void);
void lexer_start(char *filename);
void lexer_end(void);

// lexer.c

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "lexer.h"
/*****
FILE *rawfile;
int line;
double float_value;
long int_value;
char id[32];
char collected[32];
/*****
token what_collected(char* collected)
{
    char temp[32];

    strcpy(temp,collected);
    for (unsigned int i=0; i<strlen(collected); i++) collected[i] = toupper(collected[i]);
    if (strcmp(collected,"ALL")==0) { printf(" all "); return(ALL); }
    if (strcmp(collected,"EXISTS")==0) { printf(" exists ");return(EXISTS); }
    if (strcmp(collected,"OR")==0) { printf(" or "); return(OR); }
    if (strcmp(collected,"AND")==0) { printf(" and "); return(AND); }
    if (strcmp(collected,"NOT")==0) { printf(" not "); return(NOT); }
    if (strcmp(collected,"IMPLIES")==0) { printf(" implies "); return(IMPLIES); }
    if (strcmp(collected,"(")==0) { printf("("); return(LEFTP); }
    if (strcmp(collected,")")==0) { printf(")"); return(RIGHTP); }
    if (strcmp(collected,",")==0) { printf(","); return(COMMA); }
    strcpy(id,temp);
    printf("id(%s)",id);
    return(ID);
}
/*****
void fail(int line, char* message)
{
    printf("\n%s : %d\n",message,line);
    fclose(rawfile);
    exit(0);
}
/*****
int isplusalpha(char ch)
{
    if (isalpha(ch) || (ch == '-') || (ch == '_')) return(1);
    return(0);
}
/*****
token _lexer(void)
{

```

```

char ch;
int state = 0;
char chstr[2];
token result;

strcpy(collected,"");
while(1)
{
    ch = getc(rawfile);
    chstr[0] = ch;
    chstr[1] = '\0';
    if (feof(rawfile)) return(EOFFILE);
    switch(state)
    {
    case 0:
        if (ch=='-')
            { strcat(collected,chstr); state = 5; break; }
        else { ungetc(ch,rawfile); state = 1; break; }
    case 1 :
        if (isplusalpha(ch))
            { strcat(collected,chstr); state = 2; break; }
        else { ungetc(ch,rawfile); state = 3; break; }
    case 2 :
        if (isplusalpha(ch))
            { strcat(collected,chstr); break; }
        else { ungetc(ch,rawfile); result=what_collected(collected); return(result); };
    case 3 :
        if ((ch== ' ') || (ch == '\t') || (ch == '\n'))
            { if (ch == '\n') line++; state = 0; break; }
        else { ungetc(ch,rawfile); state = 4; break; }
    case 4 :
        if (ch== '#')
            { line++; while(!(getc(rawfile)=='\n')); state = 0; break; }
        else { ungetc(ch,rawfile); state = 5; break; }
    case 5 :
        if ((ch== '(') || (ch== ')') || (ch== ','))
            { strcat(collected,chstr); result=what_collected(collected); return(result); }
        else fail(line,"Invalid Character");
    default : fail(line,"Invalid Character");
    }
}
}
/*****/
void lexer_start(char *filename)
{
    if ((rawfile=fopen(filename, "rt")) == NULL) fail(0, strcat("File Missing ",filename));
    line = 0;
}
/*****/
void lexer_end(void)
{
    fclose(rawfile);
}
/*****/
token lexer(void)
{
    token result = _lexer();
    return(result);
}
/*****/

// parser.c

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include "Lexer.h"
/*****/
extern char id[32];
token tkn;
/*****/
int id_list(node **nd,node **end)
{

```

```

node *temp;

if (tkn != ID) return(INVALID);
temp = createnode(tkn,id);
if (*end != NULL)
{
    (*end)->left = temp;
    (*end)->right = NULL;
    temp->parent = *end;
}
else *nd = temp;
*end = temp;
tkn = lexer();
if (tkn == COMMA) { tkn = lexer(); return(id_list(nd,end)); }
(*end)->fullstop = true;
return(VALID);
}
/*****/
int qualifier_list(node **nd,node **end)
{
    node *temp;

    switch(tkn)
    {
    case ALL      :
    case EXISTS   :
        temp = createnode(tkn,NULL);
        temp->qual = tkn;
        if (*end != NULL)
        {
            (*end)->left = temp;
            (*end)->right = NULL;
            temp->parent = *end;
        }
        else *nd = temp;
        *end = temp;
        tkn = lexer();
        break;
    default      :
        return(VALID);
    }
    if (tkn != ID) return(INVALID);
    else
    {
        temp->tkn = tkn;
        strcpy(temp->id,id);
        tkn = lexer();
        if (qualifier_list(nd,end)) return(INVALID);
    }
    return(VALID);
}
/*****/
int term(node **nd)
{
    node *qualnd = NULL;
    node *qualend = NULL;
    node *idnd = NULL;
    node *idend = NULL;
    node *notnd = NULL;
    node *predicatend = NULL;

    if (tkn == LEFTP)
    {
        tkn = lexer();
        if (clause(nd)) return(INVALID);
        if (tkn == RIGHTP) { tkn = lexer(); return(VALID); }
        return(INVALID);
    }
    if (qualifier_list(&qualnd,&qualend)) return(INVALID);
    if (tkn == LEFTP)
    {
        tkn = lexer();
        if (clause(nd)) return(INVALID);
        if (tkn == RIGHTP)
        {
            if (qualnd != NULL)
            {

```

```

        qualend->left = *nd;
        qualend->right = NULL;
        (*nd)->parent = qualend;
        *nd = qualnd;
    }
    tkn = lexer();
    return(VALID);
}
return(INVALID);
}
if (tkn == ID)
{
    predicatend = createnode(tkn,id);
    if (isupper(id[0])) predicatend->qual = PREDICATE;
    else predicatend->qual = FUNCTION;
    tkn = lexer();
    if (tkn == LEFTP)
    {
        tkn = lexer();
        if (id_list(&idnd,&idend)) return(INVALID);
        predicatend->left = idnd;
        predicatend->end = idend;
        if (idnd == NULL) predicatend->fullstop = true;
        else idnd->parent = predicatend;
        if (tkn == RIGHTP)
        {
            tkn = lexer();
            if (qualnd != NULL)
            {
                qualend->left = predicatend;
                qualend->right = NULL;
                predicatend->parent = qualend;
                *nd = qualnd;
            }
            else *nd = predicatend;
            return(VALID);
        }
        return(INVALID);
    }
    else
    {
        if (predicatend->qual == FUNCTION) predicatend->qual = CONSTANT;
        if (qualnd != NULL)
        {
            qualend->left = predicatend;
            qualend->right = NULL;
            predicatend->parent = qualend;
            *nd = qualnd;
        }
        else *nd = predicatend;
        return(VALID);
    }
}
if (tkn == NOT)
{
    notnd = createnode(tkn,NULL);
    tkn = lexer();
    if (clause(nd)) return(INVALID);
    else
    {
        notnd->left = *nd;
        notnd->right = NULL;
        (*nd)->parent = notnd;
        *nd = notnd;
        return(VALID);
    }
}
return(INVALID);
}
/*****
int tinyclause(node **nd)
{
    node *temp = NULL;

    switch(tkn)
    {
    case OR      :

```

```

    case AND      :
    case IMPLIES:
        temp = createnode(tkn,NULL);
        temp->left = *nd;
        (*nd)->parent = temp;
        *nd = temp;
        tkn = lexer();
        break;
    default      : return(VALID);
}
temp = NULL;
if (clause(&temp)) return(INVALID);
(*nd)->right = temp;
temp->parent = *nd;
return(VALID);
}
/*****/
int clause(node **nd)
{
    if (tkn == LEFTP)
    {
        tkn = lexer();
        if (clause(nd)) return(INVALID);
        if (tkn == RIGHTP) { tkn = lexer(); return(VALID); }
        return(INVALID);
    }
    if (term(nd)) return(INVALID);
    if (tinyclause(nd)) return(INVALID);
    return(VALID);
}
/*****/
int parse_main(char *filename,node *nd[TREES])
{
    int i = 0;
    node *ndmain;

    do
    {
        ndmain = NULL;
        if (i == 0) tkn = lexer();
        if (tkn == EOFFILE) return(VALID);
        printf("\n\n*****d'th Parse\n");
        if (clause(&ndmain)) { nd[i++] = ndmain; return(INVALID); }
        nd[i++] = ndmain;
    } while (ndmain != NULL);
    return(VALID);
}
/*****/
int parse(char *filename,node *nd[TREES])
{
    int ret;

    lexer_start(filename);
    ret = parse_main(filename,nd);
    lexer_end();
    return(ret);
}
/*****/
node *createnode(token tkn, char *myid)
{
    node *temp = (node *)malloc(sizeof(node));
    temp->tkn = tkn;
    temp->qual = NOTHING;
    temp->fullstop = false;
    temp->not = false;
    temp->left = temp->right = temp->parent = temp->end = NULL;
    if (myid == NULL) strcpy(temp->id,"_____");
    else strcpy(temp->id,myid);
    return(temp);
}

// main.c

#include <stdio.h>
#include <string.h>

```

```

#include <malloc.h>
#include <conio.h>
#include <time.h>
#include "lexer.h"
/*****
int currrencnt = 0;
*****/
void print_node(node *mynode)
{
    token returned = mynode->tkn;

    if (ID==returned)
    {
        printf("id(%s) ",mynode->id);
        if (mynode->fullstop) printf("fullstop ");
        switch(mynode->qual)
        {
            case ALL          : printf("all "); break;
            case EXISTS       : printf("exists "); break;
            case CONSTANT     : printf("constant "); break;
            case PREDICATE    :
                if (mynode->end != NULL) printf("predicate (%s) ",mynode->end->id);
                else printf("predicate ");
                break;
            case FUNCTION     :
                if (mynode->end != NULL) printf("function (%s) ",mynode->end->id);
                else printf("function ");
                break;
        }
    }
    if (OR==returned) printf("or ");
    if (AND==returned) printf("and ");
    if ((NOT==returned) || mynode->not) printf("not ");
    if (IMPLIES==returned) printf("implies ");
}
/*****
node *traverse_tree(node *nd, bool print, node *op[TREES], int whichop,
                    int *opno, char *variable, char constant)
{
    node *tempnd = NULL, *left = NULL, *right = NULL;

    if (nd == NULL) return(NULL);
    if (print) { print_node(nd); printf("\n"); }
    switch(whichop)
    {
        case QUANTIFIERS: if ((nd->qual==EXISTS)||(nd->qual==ALL)) {op[*opno]=nd;(*opno)++;} break;
        case PREDICATES : if (nd->qual == PREDICATE) { op[*opno] = nd; (*opno)++; } break;
        case _IMPLIES   : if (nd->tkn == IMPLIES) { op[*opno] = nd; (*opno)++; } break;
        case _NOT       : if (nd->tkn == NOT) { op[*opno] = nd; (*opno)++; } break;
        case FINDOR     : if (nd->tkn == OR) { op[*opno] = nd; (*opno)++; } break;
        case REMOVENOT  :
            if (nd->qual == PREDICATE) nd->not = !(nd->not); else
            if (nd->qual == EXISTS) nd->qual = ALL; else
            if (nd->qual == ALL) nd->qual = EXISTS; else
            if (nd->tkn == OR) nd->tkn = AND; else
            if (nd->tkn == AND) nd->tkn = OR;
            break;
        case RENAME     :
            if ((strcmp(variable,nd->id) == 0) && (nd->qual == NOTHING))
                for(int i=0;i<currrencnt;i++) strcat(nd->id,"_");
            break;
        case REPLACECONST :
            if ((strcmp(variable,nd->id) == 0) && (nd->qual == NOTHING))
                { nd->id[0] = constant; nd->id[1] = 0; }
            break;
        case REPLACEFUNC  :
            if ((strcmp(variable,nd->id) == 0) && (nd->qual == NOTHING))
                { op[*opno] = nd; (*opno)++; }
            break;
        case TREECOPY    :
            tempnd = createnode(nd->tkn,nd->id);
            tempnd->qual = nd->qual;
            tempnd->fullstop = nd->fullstop;
            tempnd->not = nd->not;
            tempnd->end = nd->end;
            break;
    }
}

```

```

if (nd->left != NULL) left = traverse_tree(nd->left,print,op,whichop,opno,variable,constant);
if (nd->right != NULL) right = traverse_tree(nd->right,print,op,whichop,opno,variable,constant);
if (whichop == FREE) free(nd);
if (tempnd != NULL)
{
    tempnd->left = left;
    tempnd->right = right;
    if (left != NULL) left->parent = tempnd;
    if (right != NULL) right->parent = tempnd;
}
return(tempnd);
}
/*****
node *prequal(node *prend, node *op)
{
    if (op->parent != NULL)
        if (op->parent->left == op) op->parent->left = op->left;
        else op->parent->right = op->left;
    op->left->parent = op->parent;
    op->parent = prend->parent;
    if (prend->parent != NULL)
        if (prend->parent->left == prend) prend->parent->left = op;
        else prend->parent->right = op;
    prend->parent = op;
    op->left = prend;
    return(op);
}
/*****
node *prenex(node *nd)
{
    node *ops[TREES], *tempnd, *temp1, *temp2, *tnd;
    bool opsbool[TREES];
    char variable[TREES];
    int tno;

    //remove IMPLIES
    int opno = 0;
    traverse_tree(nd,false,ops,_IMPLIES,&opno,NULL,NULL);
    for(int i=0;i<opno;i++)
    {
        ops[i]->tkn = OR;
        tnd = createnode(NOT,NULL);
        tnd->left = ops[i]->left;
        ops[i]->left->parent = tnd;
        tnd->parent = ops[i];
        ops[i]->left = tnd;
    }
    //interpret NOTs
    opno = 0;
    traverse_tree(nd,false,ops,_NOT,&opno,NULL,NULL);
    for(i=opno-1;i>=0;i--)
    {
        traverse_tree(ops[i],false,NULL,REMOVENOT,NULL,NULL,NULL);
        if (ops[i]->parent != NULL)
            if (ops[i]->parent->left == ops[i]) ops[i]->parent->left = ops[i]->left;
            else ops[i]->parent->right = ops[i]->left;
        else nd = ops[i]->left;
        ops[i]->left->parent = ops[i]->parent;
        free(ops[i]);
    }
    //do the renaming and such
mylabel:
    opno = 0;
    traverse_tree(nd,false,ops,QUANTIFIERS,&opno,NULL,NULL);
    for(i=0;i<opno;i++) opsbool[i] = false;
    for(i=0;i<opno;i++)
        for(int j=i+1;j<opno;j++)
            if (strcmp(ops[i]->id,ops[j]->id) == 0)
            {
                temp1 = ops[i];
                while (temp1 != NULL)
                    if (temp1 == ops[j]) break;
                    else temp1 = temp1->parent;
                if (temp1 == NULL)
                {
                    temp2 = ops[j];
                    while (temp2 != NULL)

```

```

        if (temp2 == ops[i]) break;
        else temp2 = temp2->parent;
    }
    if ((temp1 != NULL) || (temp2 != NULL))
    {
        if (ops[i]->qual != ops[j]->qual) return(NULL);
        if (temp1 == NULL) { tnd = ops[i]; tno = i; }
        else if (temp2 == NULL) { tnd = ops[j]; tno = j; }
        else return(false);
        if (tnd->parent != NULL)
            if (tnd->parent->left == tnd) tnd->parent->left = tnd->left;
            else tnd->parent->right = tnd->left;
        tnd->left->parent = tnd->parent;
        opsbool[tno] = true;
    }
    else
    {
        if (ops[i]->qual == ops[j]->qual)
        {
            temp1 = ops[i];
            while (!(temp1->tkn == OR) || (temp1->tkn == AND) || (temp1 ==
NULL)))
                temp1 = temp1->parent;
            if (temp1 != NULL)
            {
                temp2 = ops[j];
                while (!(temp2->tkn == OR) || (temp2->tkn == AND) ||
(temp2 == NULL))
                    temp2 = temp2->parent;
                if ((temp2 != NULL) && (temp1 == temp2))
                    if (((temp1->tkn == OR) && (ops[j]->qual ==
EXISTS)) ||
                        ((temp1->tkn == AND) && (ops[j]->qual ==
ALL)))
                    {
                        if (ops[j]->parent != NULL)
                            if (ops[j]->parent->left ==
ops[j]) ops[j]->parent->left = ops[j]->left;
                            else ops[j]->parent->right =
ops[j]->left;
                        ops[j]->left->parent = ops[j]->parent;
                        free(ops[j]);
                        goto mylabel;
                    }
            }
        }
        strcpy(variable,ops[j]->id);
        currrentcnt++;
        traverse_tree(ops[j],false,NULL,RENAME,NULL,variable,NULL);
    }
}
for(i=0;i<opno;i++) if(opsbool[i]) free(ops[i]);
// carry qualifiers to prenex
tempnd = nd;
bool flg = false;
for(i=0;i<opno;i++)
{
    if (ops[i] == tempnd) { flg = true; tempnd = tempnd->left; }
    else if (!flg) nd = prequal(tempnd, ops[i]);
    else prequal(tempnd, ops[i]);
    flg = true;
}
return(nd);
}
/*****/
node *skolem(node *nd)
{
    node *tempnd = nd;
    node *freend, *startnd;
    int constcnt = 0;
    int funcnt = 0;
    int idno = 0;
    char ids[TREES][32];
    char temp[32];
    node *ops[TREES];
    int opno = 0;

```

```

while ((tempnd->qual == EXISTS) || (tempnd->qual == ALL))
{
    switch(tempnd->qual)
    {
        case EXISTS :
            if (idno == 0)
            {
                traverse_tree(nd,false,NULL,REPLACECONST,NULL,tempnd->id,'a'+constcnt);
                constcnt++;
            }
            else
            {
                traverse_tree(nd,false,ops,REPLACEFUNC,&opno,tempnd->id,NULL);
                for(int j=0;j<opno;j++)
                {
                    temp[0] = 'f'+funcnt;
                    temp[1] = 0;
                    freend = startnd = createnode(ID,temp);
                    freend->qual = FUNCTION;
                    for(int i=0;i<idno;i++)
                    {
                        freend->left = createnode(ID,ids[i]);
                        freend->left->parent = freend;
                        freend = freend->left;
                        if (i == idno-1)
                        {
                            freend->fullstop = true;
                            startnd->end = freend;
                        }
                    }
                    freend->left = ops[j]->left;
                    if (ops[j]->left != NULL) ops[j]->left->parent = freend;
                    // equalize(ops[j],startnd);
                    //*****
                    ops[j]->tkn = startnd->tkn;
                    ops[j]->qual = startnd->qual;
                    ops[j]->fullstop = startnd->fullstop;
                    ops[j]->not = startnd->not;
                    ops[j]->left = startnd->left;
                    ops[j]->right = startnd->right;
                    ops[j]->end = startnd->end;
                    strcpy(ops[j]->id,startnd->id);
                    //*****
                    startnd->left->parent = ops[j];
                    free(startnd);
                }
                funcnt++;
            }
            break;
        case ALL :
            strcpy(ids[idno],tempnd->id);
            idno++;
            break;
    }
    tempnd->left->parent = NULL;
    freend = tempnd;
    tempnd = tempnd->left;
    nd = tempnd;
    free(freend);
}
return(nd);
}
/*****
node *findornode(node *nd)
{
    node *ops[TREES];
    int opno = 0;

    traverse_tree(nd,false,ops,FINDOR,&opno,NULL,NULL);
    for(int j=0;j<opno;j++)
        if ((ops[j]->left->tkn == AND) || (ops[j]->right->tkn == AND)) return(ops[j]);
    return(NULL);
}
/*****
node *createandortree(void)
{
    node *temp = createnode(AND,NULL);
}

```

```

temp->left = createnode(AND,NULL);
temp->right = createnode(AND,NULL);
temp->left->parent = temp->right->parent = temp;
temp->left->left = createnode(OR,NULL);
temp->left->right = createnode(OR,NULL);
temp->left->left->parent = temp->left->right->parent = temp->left;
temp->right->left = createnode(OR,NULL);
temp->right->right = createnode(OR,NULL);
temp->right->left->parent = temp->right->right->parent = temp->right;
return(temp);
}
/*****/
node *copy(node *tcopy, direction dir)
{
    node *top;

    if (tcopy == NULL) return(NULL);
    top = createnode(tcopy->tkn,tcopy->id);
    top->qual = tcopy->qual;
    top->fullstop = tcopy->fullstop;
    top->not = tcopy->not;
    top->end = tcopy->end;
    switch(dir)
    {
    case left      :
        top->left = traverse_tree(tcopy->left,false,NULL,TREECOPY,NULL,NULL,NULL);
        if (top->left != NULL) top->left->parent = top;
        break;
    case right     :
        top->right = traverse_tree(tcopy->right,false,NULL,TREECOPY,NULL,NULL,NULL);
        if (top->right != NULL) top->right->parent = top;
        break;
    case both      :
        top->left = traverse_tree(tcopy->left,false,NULL,TREECOPY,NULL,NULL,NULL);
        top->right = traverse_tree(tcopy->right,false,NULL,TREECOPY,NULL,NULL,NULL);
        if (top->left != NULL) top->left->parent = top;
        if (top->right != NULL) top->right->parent = top;
        break;
    }
    return(top);
}
/*****/
void conjunct(node *nd)
{
    node *temp, *newtree, *andortree;

    if ((nd->left->tkn == AND) && (nd->right->tkn == AND))
    {
        andortree = createandortree();
        andortree->parent = nd->parent;
        if (nd->parent != NULL)
            if (nd->parent->left == nd) nd->parent->left = andortree;
            else nd->parent->right = andortree;
        andortree->left->left->left = nd->left->left;
        andortree->left->left->left->parent = andortree->left->left;
        andortree->left->left->right = nd->right->left;
        andortree->left->left->right->parent = andortree->left->left;
        andortree->left->right->left = copy(nd->left->left,both);
        andortree->left->right->left->parent = andortree->left->right;
        andortree->left->right->right = nd->right->right;
        andortree->left->right->right->parent = andortree->left->right;
        andortree->right->left->left = nd->left->right;
        andortree->right->left->left->parent = andortree->right->left;
        andortree->right->left->right = copy(nd->right->left,both);
        andortree->right->left->right->parent = andortree->right->left;
        andortree->right->right->left = copy(nd->left->right,both);
        andortree->right->right->left->parent = andortree->right->right;
        andortree->right->right->right = copy(nd->right->right,both);
        andortree->right->right->right->parent = andortree->right->right;
        free(nd->left);
        free(nd->right);
        free(nd);
    }
    else if (nd->left->tkn == AND)
    {
        newtree = copy(nd,right);
    }
}

```

```

temp = nd->left;
if (nd->parent != NULL)
    if (nd->parent->left == nd) nd->parent->left = temp;
    else nd->parent->right = temp;
temp->parent = nd->parent;
nd->left = temp->left;
temp->left->parent = nd;
temp->left = nd;
nd->parent = temp;
newtree->left = temp->right;
temp->right->parent = newtree;
temp->right = newtree;
newtree->parent = temp;
}
else if (nd->right->tkn == AND)
{
    newtree = copy(nd,left);
    temp = nd->right;
    if (nd->parent != NULL)
        if (nd->parent->left == nd) nd->parent->left = temp;
        else nd->parent->right = temp;
    temp->parent = nd->parent;
    nd->right = temp->right;
    temp->right->parent = nd;
    temp->right = nd;
    nd->parent = temp;
    newtree->right = temp->left;
    temp->left->parent = newtree;
    temp->left = newtree;
    newtree->parent = temp;
}
}
/*****
node *conjunctive(node *nd)
{
    node *newnode;

    newnode = findornode(nd);
    while (newnode != NULL) { conjunct(newnode); newnode = findornode(nd); }
    return(nd);
}
*****/
bool minimize(node *nd[TREES],int *treec)
{
    for(int i=0;i<(*treec);i++)
        if (nd[i]->tkn == AND)
            {
                node *temp = nd[i];
                nd[*treec] = temp->right;
                nd[i] = temp->left;
                free(temp);
                nd[i]->parent = nd[*treec]->parent = NULL;
                (*treec)++;
                return(true);
            }
    return(false);
}
/*****
node *unifyandresolve(node *first, node *second, node *rf, node *rs, substitution *unifier)
{
    node *opfirst[TREES],*opsecond[TREES],*tempnd,*temp;
    int opnofirst,opnosecond,i;
    substitution *tempsub;

    if (!first || !second || !rf || !rs || !unifier) return(NULL);
    tempsub = unifier;
    while((tempsub != NULL) && !tempsub->dummy)
    {
        opnofirst = opnosecond = 0;
        traverse_tree(first,false,opfirst,REPLACEFUNC,&opnofirst,tempsub->variable->id,NULL);
        traverse_tree(second,false,opsecond,REPLACEFUNC,&opnosecond,tempsub->variable->id,NULL);
        switch(tempsub->stable->qual)
        {
            case CONSTANT :
            case NOTHING :
                for(i=0;i<opnofirst;i++) strcpy(opfirst[i]->id,tempsub->stable->id);
                for(i=0;i<opnosecond;i++) strcpy(opsecond[i]->id,tempsub->stable->id);

```

```

        break;
case PREDICATE :
case FUNCTION :
    if (tempsub->stable->end == NULL)
    {
        tempnd = tempsub->stable->left;
        tempsub->stable->left = NULL;
    }
    else
    {
        tempnd = tempsub->stable->end->left;
        tempsub->stable->end->left = NULL;
    }
    node *newstable = traverse_tree(tempsub->stable,false,NULL,TREECOPY,NULL,NULL,NULL);
    if (tempsub->stable->end == NULL) tempsub->stable->left = tempnd;
    else tempsub->stable->end->left = tempnd;
    if (tempsub->variable->parent != NULL)
        if (tempsub->variable->parent->left == tempsub->variable)
            tempsub->variable->parent->left = newstable;
        else tempsub->variable->parent->right = newstable;
    newstable->parent = tempsub->variable->parent;
    if (newstable->left == NULL) newstable->left = tempsub->variable->left;
    else newstable->end->left = tempsub->variable->left;
    break;
}
tempsub = tempsub->next;
}
while(unifier != NULL) { tempsub = unifier->next; free(unifier); unifier = tempsub; }
if ((rf->parent == NULL) && (rs->parent == NULL)) return(NULL); // both NULL
if (rf->parent != NULL)
    if (rs->parent != NULL) // both not NULL
    {
        if (rf->parent->left == rf)
        {
            if (rs->parent->left == rs) rs->parent->left = rf->parent->right;
            else rs->parent->right = rf->parent->right;
            rf->parent->right->parent = rs->parent;
        }
        else
        {
            if (rs->parent->left == rs) rs->parent->left = rf->parent->left;
            else rs->parent->right = rf->parent->left;
            rf->parent->left->parent = rs->parent;
        }
    }
    // equalize(rf->parent,second)
    //*****
    rf->parent->tkn = second->tkn;
    rf->parent->qual = second->qual;
    rf->parent->fullstop = second->fullstop;
    rf->parent->not = second->not;
    rf->parent->left = second->left;
    rf->parent->right = second->right;
    rf->parent->end = second->end;
    strcpy(rf->parent->id,second->id);
    //*****
    second->left->parent = second->right->parent = rf->parent;
    free(second);
}
else // rs->parent NULL rf->parent not NULL
{
    if (rf->parent->left == rf) temp = rf->parent->right;
    else temp = rf->parent->left;
    // equalize(rf->parent,temp)
    //*****
    rf->parent->tkn = temp->tkn;
    rf->parent->qual = temp->qual;
    rf->parent->fullstop = temp->fullstop;
    rf->parent->not = temp->not;
    rf->parent->left = temp->left;
    rf->parent->right = temp->right;
    rf->parent->end = temp->end;
    strcpy(rf->parent->id,temp->id);
    //*****
    free(temp);
}
else // rf->parent NULL rs->parent not NULL
{

```

```

        if (rs->parent->left == rs) temp = rs->parent->right;
        else temp = rs->parent->left;
        // equalize(rf->parent,temp)
        /*****
        rs->parent->tkn = temp->tkn;
        rs->parent->qual = temp->qual;
        rs->parent->fullstop = temp->fullstop;
        rs->parent->not = temp->not;
        rs->parent->left = temp->left;
        rs->parent->right = temp->right;
        rs->parent->end = temp->end;
        strcpy(rs->parent->id,temp->id);
        /*****
        free(temp);
        first = second;
    }
    while (rf != NULL) { temp = rf; rf = rf->left; free(temp); }
    while (rs != NULL) { temp = rs; rs = rs->left; free(temp); }
    return(first);
}
/*****/
substitution *findunifier(node *first, node *second, substitution *old)
{
    substitution *temp;

    if ((first == NULL) && (second != NULL)) return(NULL);
    if ((first != NULL) && (second == NULL)) return(NULL);
    if ((first == NULL) && (second == NULL)) return(old);
    if (strcmp(first->id,second->id) == 0) return(findunifier(first->left,second->left,old));
    else
    {
        if ((first->tkn == ID) && (second->tkn == ID))
        {
            temp = (substitution *)malloc(sizeof(substitution));
            temp->dummy = false;
            temp->next = old;
            if (first->qual==NOTHING)
            {
                temp->variable = first;
                temp->stable = second;
                switch(second->qual)
                {
                    case CONSTANT :
                    case NOTHING : return(findunifier(first->left,second->left,temp));
                    case PREDICATE :
                    case FUNCTION :
                        if (second->end == NULL) return(findunifier(first->left,NULL,temp));
                        return(findunifier(first->left,second->end->left,temp));
                }
            }
            if (second->qual==NOTHING)
            {
                temp->stable = first;
                temp->variable = second;
                switch(first->qual)
                {
                    case CONSTANT :
                    case NOTHING : return(findunifier(first->left,second->left,temp));
                    case PREDICATE :
                    case FUNCTION :
                        if (first->end == NULL) return(findunifier(NULL,second->left,temp));
                        return(findunifier(first->end->left,second->left,temp));
                }
            }
        }
    }
    return(NULL);
}
/*****/
bool isresolvent(node *first, node *second, node **fs, node **ss)
{
    int opnofirst = 0, opnosecond = 0;
    node *opsfirst[TREES], *opssecond[TREES];

    traverse_tree(first,false,opsfirst,PREDICATES,&opnofirst,NULL,NULL);
    traverse_tree(second,false,opssecond,PREDICATES,&opnosecond,NULL,NULL);
    for(int i=0;i<opnofirst;i++)

```

```

        for(int j=0;j<opnosecond;j++)
            if ((strcmp(opsfirst[i]->id,opssecond[j]->id) == 0) &&
                (opsfirst[i]->not ^ opssecond[j]->not))
                { *fs = opsfirst[i]; *ss = opssecond[j]; return(true); }
    return(false);
}
/*****/
bool sametree(node *first, node *second)
{
    if ((first == NULL) && (second == NULL)) return(true);
    if ((first == NULL) || (second == NULL)) return(false);
    if ((first->tkn != second->tkn) || (first->fullstop != second->fullstop) ||
        (first->qual != second->qual) || (first->not != second->not) ||
        (strcmp(first->id,second->id)))
        return(false);
    if (!sametree(first->left,second->left)) return(false);
    if (!sametree(first->right,second->right)) return(false);
    return(true);
}
/*****/
bool inarray(node *nd[TREES], node *resolvent, int treec)
{
    for(int i=0;i<treec;i++) if (sametree(nd[i],resolvent)) return(true);
    return(false);
}
/*****/
void printalltrees(node *nd[TREES],char *Message)
{
    for(int i=0;i<TREES;i++)
    {
        if (nd[i] == NULL) break;
        printf("\n*****%d'th Tree Traversal - %s\n",i,Message);
        traverse_tree(nd[i],true,NULL,NONE,NULL,NULL,NULL);
        printf("*****\n");
    }
}
/*****/
bool resolve(node *nd[TREES], int *treec)
{
    int i,j;
    substitution *unifier;
    node *resolvent, *ri, *rj;

start:
    for(i=0;i<(*treec);i++)
        for(j=i+1;j<(*treec);j++)
        {
            node *first = copy(nd[i],both);
            node *second = copy(nd[j],both);
            if (isresolvent(first,second,&ri,&rj))
            {
                substitution *temp = (substitution *)malloc(sizeof(substitution));
                temp->dummy = true;
                temp->next = NULL;
                unifier = findunifier(ri,rj,temp);
                if (unifier != NULL)
                {
                    resolvent = unifyandresolve(first,second,ri,rj,unifier);
                    if ((resolvent != NULL) && (!inarray(nd,resolvent,*treec)))
                    {
                        nd[*treec] = resolvent;
                        printf("\n*****Tree %d After Resolution %d with
%d\n",*treec,i,j);

                        traverse_tree(nd[*treec],true,NULL,NONE,NULL,NULL,NULL);
                        printf("*****\n");
                        (*treec)++;
                        goto start;
                    }
                    else if (resolvent == NULL)
                    {
                        printf("\n*****Contradiction Detected in %d with %d\n",i,j);
                        return(false); //contradiction
                    }
                }
            }
            else return(false);
        }
    }
}
else

```

```

        {
            traverse_tree(first, false, NULL, FREE, NULL, NULL, NULL);
            traverse_tree(second, false, NULL, FREE, NULL, NULL, NULL);
        }
    }
    return(true);
}
/*****
void main(void)
{
    node *nd[TREES];
    char tmpbuf[128];

    for(int i=0;i<TREES;i++) nd[i] = 0;
    printf("\n-----\n");
    _strtime(tmpbuf); printf("OS Time: %s\n", tmpbuf);
    _strdate(tmpbuf); printf("OS Date: %s\n\n", tmpbuf);
//-----
    if (parse("Sample.Txt",nd)) { printf("\n***** Parse Error\n\n"); return; }
    else printf("\n***** Parse Ok, Now Tree Check\n");
    printalltrees(nd,"Normal Traversal");
//-----
    for(i=0;i<TREES;i++)
    {
        if (nd[i] == NULL) break; else nd[i] = prenex(nd[i]);
        if (nd[i] == NULL) { printf("\n*****Problem in Prenex\n"); return; }
    }
    printalltrees(nd,"After Prenex");
//-----
    for(i=0;i<TREES;i++)
    {
        if (nd[i] == NULL) break; else nd[i] = conjunctive(nd[i]);
        if (nd[i] == NULL) { printf("\n*****Problem in Conjunctive\n"); return; }
    }
    printalltrees(nd,"After Conjunctive");
//-----
    for(i=0;i<TREES;i++)
    {
        if (nd[i] == NULL) break; else nd[i] = skolem(nd[i]);
        if (nd[i] == NULL) { printf("\n*****Problem in Skolem\n"); return; }
    }
    printalltrees(nd,"After Skolem");
//-----
    for(int treec=0;treec<TREES;treec++) if (nd[treec] == NULL) break;
    while (minimize(nd,&treec));
    printalltrees(nd,"After Minimize");
//-----
    for(treec=0;treec<TREES;treec++) if (nd[treec] == NULL) break;
    if (resolve(nd,&treec)) printalltrees(nd,"Resolution Ok - Satisfiable");
    else printf("\n*****Resolution Nok - Unsatisfiable\n");
//-----
    printf("\n-----\n");
}
*****/

```