G C Code

This appendix contains source code listings of all the data structures, diagrammer code, and probench .pr0 file handling code. Where necessary the listings are preceded by text to identify them and their use more explicitly.

G.1 Interpreter Source Code

G.1.1 Symbol Table

```
/*==========================================================*
 *                SYMTAB.H                                  *
 * the symbol table structure and the function definitions *
 *==========================================================*/
#ifndef SYMTABY
#define SYMTABY

#include "node.h"

/* arbitrary size of symbol table */

#define SYMTAB_ARRAY_SIZE 100

/* the symbol table entry structure consists of a list
   of node_lists, meaning that a number of variables
   can share the same hash value, but also that variables
   can share the same name */

typedef struct{
                NODE_LIST *contents;
                struct SYMTAB_ENTRY *next;
             }SYMTAB_ENTRY;

/********** functions ************/
extern SYMTAB_ENTRY **create_symbol_table_array(int);
extern void add_node_to_sym(NODE *,int,SYMTAB_ENTRY **);
extern SYMTAB_ENTRY *create_symtab_entry(NODE_LIST *);
extern int hash(char *);
extern void print_nodes(SYMTAB_ENTRY **);
extern void print_node(NODE *);
extern void save_nodes(SYMTAB_ENTRY **,FILE *);
extern void save_node(NODE *,FILE *);
extern int load_symtab(SYMTAB_ENTRY **,char *);
extern NODE *look_for(int,int,SYMTAB_ENTRY **);
extern NODE *look_for_by_name(char *,SYMTAB_ENTRY **);
```

```
extern void cross_reference(SYMTAB_ENTRY **,int);
extern void quick_display_symtab(SYMTAB_ENTRY **,int);
extern void quick_display_node(NODE *);
extern void reset_symtab(SYMTAB_ENTRY **,int);
extern void reset_node(NODE *);


#endif
```

```
/*=================================================*
 *                SYMTAB.C                         *
 * functions for the symbol table in the interpreter *
 *=================================================*/
#include "symtab.h"
#include "alfile.h"
#include "alstring.h"
#include "errors.h"

/*=================================================================*
 * create_symtab_entry , allocates enough space for a SYMTAB_ENTRY *
 *=================================================================*/
SYMTAB_ENTRY *create_symtab_entry(NODE_LIST *nlp)
   {
     SYMTAB_ENTRY *sep=(SYMTAB_ENTRY *)malloc(sizeof(SYMTAB_ENTRY));

                if(sep==NULL)
                        fatal_error("NO room for SYMTAB_ENTRY");

                sep->contents=nlp;
                sep->next=NULL;

                return(sep);
   }

/*=================================================================*
 * add node to slot, first of all checks the slot, if null then    *
 * just adds it to the slot, otherwise looks for a list containing *
 * the variable name and adds on to it, if not then it creates a   *
 * new node list                                                   *
 *=================================================================*/
void add_node_to_sym(NODE *node,int slot,SYMTAB_ENTRY *array[])
   {
     SYMTAB_ENTRY *sep;
     NODE_LIST *nlp;

     if(array[slot]!=NULL)
         {
```

```
        if(nlp=extract_link(NAME(node),array[slot]->contents)==NULL)
            {
              /* couldn't find it*/
              nlp=array[slot]->contents;

              while(nlp->link!=NULL)
                 nlp=nlp->link;

              nlp->link=create_node_list(node);
            }
        else
            {
              sep=array[slot];
              while(sep->next!=NULL)
                  sep=sep->next;

              sep->next=create_symtab_entry(
                              create_node_list(node));
            }
      }
    else
        array[slot]=create_symtab_entry(create_node_list(node));
}

/*=================================================*
 * a completely randomly written hash function      *
 * with no thought given to efficient distribution of *
 * variables                                        *
 *=================================================*/
int hash(char *hash_string)
      {

            int len=strlen(hash_string);
            int n=0,loop;

      /* works through string, increasing the value
         of n dependant upon the current value of n
         in relation to loop, and the current character */
```

```
                for(loop=0;loop<len;loop++){
                        n+=((n>loop)?n-loop:n+loop)*(hash_string[loop]);
                        }

                if(n<0)n=n*(-1);

                return(n);
        }

/*=========================================================*
 *  create_symbol_table_array allocates space for array *
 *=========================================================*/
SYMTAB_ENTRY **create_symbol_table_array(int size)
        {
                int loop;
                SYMTAB_ENTRY **stp=(SYMTAB_ENTRY **)
                            malloc(sizeof(SYMTAB_ENTRY *)*size);

                if(stp==NULL)
                        fatal_error("no more room for symbol table ARRAY");

                for(loop=0;loop<size;loop++)
                        stp[loop]=NULL;

                return(stp);
        }

/*==============================================================*
 * a test function used to test the save but left in because *
 * it provides some useful information for debugging         *
 * it works through the array, and every list in the array   *
 * printing off each node                                    *
 *==============================================================*/
void print_nodes(SYMTAB_ENTRY **symtab)
   {
     int slot=0;
     SYMTAB_ENTRY *sep;
     NODE_LIST *nlp;
```

```
    for(slot=0;slot<100;slot++)
        {
         sep=symtab[slot];
         if(sep!=NULL)
             {
              while(sep->next!=NULL)
                 {
                  nlp=sep->contents;
                  while(nlp->link!=NULL)
                     {
                       print_node(nlp->node);
                       nlp=nlp->link;
                     }
                  print_node(nlp->node);
                  sep=sep->next;
                 }
             nlp=sep->contents;
             while(nlp->link!=NULL)
                 {
                   print_node(nlp->node);
                   nlp=nlp->link;
                 }
             print_node(nlp->node);
            }
        else printf("printing slot %d is empty --",slot);
        }
    printf("END of NODES\n");
  }

void print_node(NODE *node)
  {
    NODE *np;

    printf("own hash = %d\n",node->own.hash);
    printf("own uniq = %d\n",node->own.unique);
    np=node->parent;
    if(np==NULL)
       {
```

```
                printf("\n\nparent hash = 0\n");
                printf("parent uniq = 0\n");
        }
      else{
                printf("\n\nparent hash = %d\n",np->own.hash);
                printf("parent uniq = %d\n",np->own.unique);
        }
      np=node->younger_sibling;
      if(np==NULL)
        {
                printf("sibling hash = 0\n");
                printf("parent uniq = 0\n");
        }
      else{

                printf("sibling hash = %d\n",np->own.hash);
                printf("sibling uniq = %d\n",np->own.unique);
        }
       np=node->eldest_child;
       if(np==NULL)
        {

                printf("child hash = 0\n");
                printf("child uniq = 0\n");
        }
      else{

            printf("child hash = %d\n",np->own.hash);
            printf("child uniq = %d\n",np->own.unique);
        }

                printf("level  %d\n",node->level);

                printf("\nDATA INFO\n");
                printf("NAME %s\n",INFO(node)->name);
                printf("TYPE %d\n",INFO(node)->type);
if(INFO(node)->type==INT)
   printf("integer value is data_n  %d\n",INFO(node)->data_n);
else{
```

```
      if(INFO(node)->type==CHAR)
        {
                  printf("data_c %s\n",INFO(node)->data_c);
                  printf("size data_n %d\n",INFO(node)->data_n);
        }
      }
getchar();
 }

/*==========================================================*
 * save_nodes saves the symbol table to the specified file *
 *==========================================================*/
void save_nodes(SYMTAB_ENTRY **symtab,FILE *outfile)
  {
     int slot=0;
     SYMTAB_ENTRY *sep;
     NODE_LIST *nlp;


     for(slot=0;slot<100;slot++) /* loop through slots */
         {
           sep=symtab[slot];
           if(sep!=NULL)       /* if there are entries */
                {
                 while(sep->next!=NULL) /* loop through list */
                   {
                     nlp=sep->contents;
                     while(nlp->link!=NULL) /* loop through list */
                        {
                         save_node(nlp->node,outfile);
                         nlp=nlp->link;
                        }
                     save_node(nlp->node,outfile);
                     sep=sep->next;
                   }
                nlp=sep->contents;      /* do last list */
                while(nlp->link!=NULL)
                   {
                     save_node(nlp->node,outfile);
```

```
                    nlp=nlp->link;
                   }
                save_node(nlp->node,outfile);
                }
               }
            fprintf(outfile,"END of NODES\n");
      }

/*=========================================*
 * save_node writes out a node to the file *
 *=========================================*/
void save_node(NODE *node,FILE *outfile)
        {
                NODE *np;

        /* write header with the hash, and unique number */
        fprintf(outfile,"NODE\N");
                fprintf(outfile,"%d\n",node->own.hash);
                fprintf(outfile,"%d\n",node->own.unique);


        /* for each of the parent, child and sibling
           if none exist then write out null values,
           otherwise write out the hash and unique
           number of the relative */

        np=node->parent;
        if(np==NULL)
          {
                fprintf(outfile,"0\n");
                fprintf(outfile,"0\n");
          }
        else{
                    fprintf(outfile,"%d\n",np->own.hash);
                    fprintf(outfile,"%d\n",np->own.unique);
            }

        np=node->younger_sibling;
        if(np==NULL)
```

```
        {
                fprintf(outfile,"0\n");
                fprintf(outfile,"0\n");
        }
      else{
              fprintf(outfile,"%d\n",np->own.hash);
                fprintf(outfile,"%d\n",np->own.unique);
        }

      np=node->eldest_child;
      if(np==NULL)
        {
                fprintf(outfile,"0\n");
                fprintf(outfile,"0\n");
        }
      else{
              fprintf(outfile,"%d\n",np->own.hash);
              fprintf(outfile,"%d\n",np->own.unique);
        }

      /* write out the level number */
              fprintf(outfile,"%d\n",node->level);


      /* write out the variable name and type */
              fprintf(outfile,"%s\n",INFO(node)->name);
              fprintf(outfile,"%d\n",INFO(node)->type);

/* if its an int then write out the contents,
   if its a char then write out the string length
   followed by the string */

if(INFO(node)->type==INT)
   fprintf(outfile,"%d\n",INFO(node)->data_n);
else{
    if(INFO(node)->type==CHAR)
      {
              fprintf(outfile,"%d\n",INFO(node)->data_n);
              fprintf(outfile,"%s\n",INFO(node)->data_c);
```

```
        }
      }
  }

/*=================================================*
 * load_symtab loads in a symbol table from filename *
 *=================================================*/
int load_symtab(SYMTAB_ENTRY **sym,char *filename)
{
     FILE *infile=open_input(filename);
     char header[20],g,*cp,name[100];
     NODE *current;
     int loop=0;


     fgets(header,20,infile);

     while(header[0]=='N')
       {
        current=create_node();
        fscanf(infile,"%d",&(current->own.hash));
        g=getc(infile);
        fscanf(infile,"%d",&(current->own.unique));
        g=getc(infile);
        fscanf(infile,"%d",&(current->parent_stuff.hash));
        g=getc(infile);
        fscanf(infile,"%d",&(current->parent_stuff.unique));
        g=getc(infile);
        fscanf(infile,"%d",&(current->sibling.hash));
        g=getc(infile);
        fscanf(infile,"%d",&(current->sibling.unique));
        g=getc(infile);
        fscanf(infile,"%d",&(current->child.hash));
        g=getc(infile);
        fscanf(infile,"%d",&(current->child.unique));
        g=getc(infile);
        fscanf(infile,"%d",&(current->level));
        g=getc(infile);
/*data info*/
```

```
 /* get name */
 loop=0;
 g=getc(infile);
 while(g!='\n'){
    name[loop]=g;
    g=getc(infile);
    loop++;
   }
 name[loop]='\0';
current->info->name=(char *)malloc(loop);
 strcpy(current->info->name,name);


 fscanf(infile,"%d",&(current->info->type));
 g=getc(infile);

if(current->info->type==INT)
 {
 fscanf(infile,"%d",&(current->info->data_n));
 g=getc(infile);
 current->info->data_c=NULL;
 }
else
   {
   if(current->info->type==CHAR)
     {
      fscanf(infile,"%d",&(current->info->data_n));
      g=getc(infile);
      current->info->data_c=(char *)malloc(current->info->data_n + 1);
      if(current->info->data_c==NULL)
        {
         printf("couldn't allocate enough memory for data string\n");
         return(0);
        }
      g=getc(infile);
      cp=current->info->data_c;
      while(g!='\n'){
         *cp=g;
         cp++;
```

```
                g=getc(infile);
               }
             *cp='\0';
           }
         }

         add_node_to_sym(current,current->own.hash,sym);
         fgets(header,20,infile);
        }

     close_file(infile);
  return(1);
}

/*=====================================================*
 * cross_reference a symbol table loops through       *
 * the table trying to match up parents with children *
 * brothers with sisters and children with parents    *
 *=====================================================*/
void cross_reference(SYMTAB_ENTRY **symtab,int size)
{
  NODE *current,*seek;
  NODE_LIST *nlp;
  SYMTAB_ENTRY *sep;
  int slot;

  for(slot=0;slot<100;slot++)
          {
           sep=symtab[slot];
           if(sep!=NULL)
              {
               while(sep->next!=NULL)
                  {
                   nlp=sep->contents;
                   while(nlp->link!=NULL)
                      {
                       current=nlp->node;
                       seek=look_for(current->parent_stuff.unique,
                                     current->parent_stuff.hash,
```

```
                    symtab);
            current->parent=seek;
            seek=look_for(current->sibling.unique,
                    current->sibling.hash,
                    symtab);
            current->younger_sibling=seek;
            seek=look_for(current->child.unique,
                    current->child.hash,
                    symtab);
            current->eldest_child=seek;
            nlp=nlp->link;
          }
        current=nlp->node;
        seek=look_for(current->parent_stuff.unique,
                current->parent_stuff.hash,
                symtab);
        current->parent=seek;
        seek=look_for(current->sibling.unique,
                current->sibling.hash,
                symtab);
        current->younger_sibling=seek;
        seek=look_for(current->child.unique,
                current->child.hash,
                symtab);
        current->eldest_child=seek;
        nlp=nlp->link;
        sep=sep->next;
      }
    nlp=sep->contents;
    while(nlp->link!=NULL)
      {
        current=nlp->node;
        seek=look_for(current->parent_stuff.unique,
                current->parent_stuff.hash,
                symtab);
        current->parent=seek;
        seek=look_for(current->sibling.unique,
                current->sibling.hash,
                symtab);
```

```
                    current->younger_sibling=seek;
                    seek=look_for(current->child.unique,
                                  current->child.hash,
                                  symtab);
                    current->eldest_child=seek;
                    nlp=nlp->link;
                  }

                  current=nlp->node;
                  seek=look_for(current->parent_stuff.unique,
                                current->parent_stuff.hash,
                                symtab);
                  current->parent=seek;
                  seek=look_for(current->sibling.unique,
                                current->sibling.hash,
                                symtab);
                  current->younger_sibling=seek;
                  seek=look_for(current->child.unique,
                                current->child.hash,
                                symtab);
                  current->eldest_child=seek;
                }
        }
}

/*=================================================*
 * look_for seeks through the hash slot for the node *
 * with the unique id number unique                  *
 *=================================================*/
NODE *look_for(int unique,int hash,SYMTAB_ENTRY **symtab)
{
   NODE *np;
   NODE_LIST *nlp;
   SYMTAB_ENTRY *sep; /*contents nodelist pointer, next sep*/

       if(symtab[hash]==NULL) /* the relative may not exist */
      {
       return NULL;
      }
```

```
    sep=symtab[hash];

    while(sep->next!=NULL) /* run through the list */
      {
       nlp=sep->contents;
       while(nlp->link!=NULL) /* run through the list */
          {
           np=nlp->node;
           if(np->own.unique==unique) /* have we found it?*/
               return(np);
           nlp=nlp->link;
          }
           np=nlp->node;
           if(np->own.unique==unique) /* found ? */
               return(np);
           sep=sep->next;
      }
       nlp=sep->contents;       /* do the last list */
       while(nlp->link!=NULL)
          {
           np=nlp->node;
           if(np->own.unique==unique)
               return(np);
           nlp=nlp->link;
          }
           np=nlp->node;
           if(np->own.unique==unique)
               return(np);

printf("ERROR 2 in RECROSS\n");
return NULL;
}

/*=====================================================*
 * look_for_by_name uses the variable name rather than *
 * hash and unique, the name is hashed to find the slot*
 * that must be checked                                 *
 *=====================================================*/
```

```
NODE *look_for_by_name(char *name,SYMTAB_ENTRY **symtab)
{
   NODE *np;
   NODE_LIST *nlp;/* node , link */
   SYMTAB_ENTRY *sep; /*contents nodelist pointer, next sep*/
   int hash_val;

   hash_val=hash(name)%100;

   if(symtab[hash_val]==NULL)
       {
        return NULL;
       }

    sep=symtab[hash_val];

    while(sep->next!=NULL)
      {
       nlp=sep->contents;
       while(nlp->link!=NULL)
          {
           np=nlp->node;
           if(!strcmp(name,np->info->name))
               return(np);
           nlp=nlp->link;
          }
           np=nlp->node;
           if(!strcmp(name,np->info->name))
               return(np);
       sep=sep->next;
      }
       nlp=sep->contents;
       while(nlp->link!=NULL)
          {
           np=nlp->node;
           if(!strcmp(name,np->info->name))
               return(np);
           nlp=nlp->link;
          }
```

```
            np=nlp->node;
            if(!strcmp(name,np->info->name))
                return(np);
printf("ERROR 2 in RECROSS\n");
return NULL;
}

/*=========================================================*
 * run through the symbol table, no ordering is achieved *
 * because the very first 01 level has not been saved    *
 * although it is recorded by the symbol table generator *
 *=========================================================*/
void quick_display_symtab(SYMTAB_ENTRY **symtab,int size)
   {
      int slot=0;
      SYMTAB_ENTRY *sep;
      NODE_LIST *nlp;

      for(slot=0;slot<size;slot++)
         {
            sep=symtab[slot];
            if(sep!=NULL)
                 {
                  while(sep->next!=NULL)
                     {
                      nlp=sep->contents;
                      while(nlp->link!=NULL)
                         {
                          quick_display_node(nlp->node);
                          nlp=nlp->link;
                         }
                      quick_display_node(nlp->node);
                      sep=sep->next;
                     }
                 nlp=sep->contents;
                 while(nlp->link!=NULL)
                    {
                      quick_display_node(nlp->node);
                      nlp=nlp->link;
```

```
                    }
                  quick_display_node(nlp->node);
                }
            }
        printf("END of SYMBOLTABLE\n");
    }

/*=================================================*
 * display the node level number, name and contents *
 *=================================================*/
void quick_display_node(NODE *node)
       {

              printf("%d   ",node->level);

              printf("- %s ",INFO(node)->name);

       if(INFO(node)->type==INT)
           printf(" =  %d\n",INFO(node)->data_n);
       else{
            if(INFO(node)->type==CHAR)
              {
               printf("data_c %s\n",INFO(node)->data_c);
              }
            else
               printf("\n");
           }
       }


/*=========================================================*
 * this is a primitive reset function to allow a trace      *
 * to begin again from the start, it makes all numerics     *
 * 0 and chars '\0', a more sophisticated mechanism would   *
 * be needed for VALUE implementations, so that the current *
 * value and the default value could be stored, then the    *
 * current could be replaced with the default               *
 *=========================================================*/
```

```
void reset_symtab(SYMTAB_ENTRY **symtab,int size)
  {
     int slot=0;
     SYMTAB_ENTRY *sep;
     NODE_LIST *nlp;

     for(slot=0;slot<size;slot++)
          {
            sep=symtab[slot];
            if(sep!=NULL)
               {
                while(sep->next!=NULL)
                  {
                   nlp=sep->contents;
                   while(nlp->link!=NULL)
                       {
                         reset_node(nlp->node);
                         nlp=nlp->link;
                       }
                   reset_node(nlp->node);
                   sep=sep->next;
                  }
                nlp=sep->contents;
                while(nlp->link!=NULL)
                   {
                    reset_node(nlp->node);
                    nlp=nlp->link;
                   }
                reset_node(nlp->node);
               }
          }
       printf("RESET SYMBOLTABLE\n");
    }

void reset_node(NODE *node)
{

  if(INFO(node)->type==INT)
       INFO(node)->data_n=0;
```

```
  else{
      if(INFO(node)->type==CHAR)
                  *(INFO(node)->data_c)='\0';
      }
}
```

```
/*=================================================*
 *                      NODE.H                     *
 * the structures used to represent variables in the *
 * interpreter                                     *
 *=================================================*/

#ifndef NODEY
#define NODEY
#include <stdio.h>

/* type of variables */
typedef enum{
             VARIABLE,
             QUALIFIER,  /* a variable name with no pic */
             STRING,
             INT,        /* numeric variable */
             CHAR        /* character variable */
            }TYPE;

/* the next structure is only for non inverted use */
/* used when saving variables to files, hash is the
   array position, unique is the value allocated to that
   variable */

typedef struct{  int hash;
                 int unique;
              }save_values;


/* the information relating to the variable
   the name
       type - char , int etc
       format - not implemented properly yet
       data_c - for char variables pointer to the string
       data_n - for ints the value, for char the length
*/

typedef struct{
                char *name;
```

```
                TYPE type;
                char *format;
                char *data_c;
                int data_n;
              }DATA_INFO;

/* parent is the working storage level above the current
   younger_sibling, is the next variable at the same level
   eldest_child the first child variable
   info is the data for the variable
   level is the WS level ie 01, 02 etc.
*/

typedef struct{
               struct NODE *parent;
               save_values parent_stuff;
               struct NODE *younger_sibling;
               save_values sibling;
               struct NODE *eldest_child;
               save_values child;
               DATA_INFO *info;
               save_values own;
               int level;
              }NODE;

/* linked list definition for nodes */

typedef struct{
                NODE *node;
                struct NODE_LIST *link;
              }NODE_LIST;

/********************
** Macro's
********************/
#define LEVEL(N)        ((N)->level)
#define INFO(N)         ((N)->info)
#define PARENT(N)       ((N)->parent)
#define SIBLING(N)      ((N)->younger_sibling)
```

```
#define CHILD(N)        ((N)->eldest_child)

/* DATA INFO MACRO'S */
#define NAME(N)         ((N)->info->name)

/* DEFINE NODE FUNCTIONS */
extern NODE_LIST *create_node_list(NODE *);
extern NODE *create_node(void);

extern NODE_LIST *extract_link(char *,NODE_LIST *);

extern DATA_INFO *create_data_info(void);

/*macro function's*/

#define GET_NODE(NL)    ((NL)->node)
#define NEXT_LIST(NL)   ((NL)->link)
#define CREATE_BLANK_NODE_LIST()        (create_node_list(NULL))

#endif
```

```
/*================================================*
 *                    NODE.C                      *
 *   the functions as used for variable nodes     *
 *================================================*/

#include "node.h"
#include "errors.h"

/*==============================================================*
 * create_data_info allocates enough space for the variable *
 * information                                              *
 *==============================================================*/
DATA_INFO *create_data_info()
        {
                DATA_INFO *dip=(DATA_INFO *)malloc(sizeof(DATA_INFO));

                if(dip==NULL)
                        fatal_error("no more room for DATA_INFO");

                dip->name=NULL;
                dip->format=NULL;
                dip->data_c=NULL;
                dip->data_n=0;
                return(dip);
        }

/*================================================*
 * create_node allocates enough space for the node *
 *================================================*/
NODE *create_node()
        {
                NODE *ndp=(NODE *)malloc(sizeof(NODE));

                if(ndp==NULL)
                        fatal_error("no more room for NODE");

                PARENT(ndp)=NULL;
                SIBLING(ndp)=NULL;
                CHILD(ndp)=NULL;
```

```
                INFO(ndp)=create_data_info();
                LEVEL(ndp)=0;

                return(ndp);
        }

/*==================================================*
 * create_node_list creates space for the node list *
 *==================================================*/
NODE_LIST *create_node_list(NODE *ndp)
        {
                NODE_LIST *ndlp=(NODE_LIST *)malloc(sizeof(NODE_LIST));

                if(ndlp==NULL)
                        fatal_error("no more room for NODE LIST");

                ndlp->node=ndp;
                ndlp->link=NULL;

                return(ndlp);
        }

/*==================================================*
 * extract_link finds a variable in a list of nodes *
 *==================================================*/
NODE_LIST *extract_link(char *name,NODE_LIST *nlp)
        {
                NODE_LIST *nlpoint=nlp;

                while(nlpoint!=NULL)
                        {
                                if(!strcmp(NAME(nlpoint->node),name))
                                        return nlpoint;
                                else
                                        nlpoint=nlpoint->link;
                        }
                return nlpoint;
        }
```

G.1.2 Operations Table

```
/*==========================================================*
 *              OPS.H                                       *
 * define's and declarations for the operations used in the *
 * interpreter                                              *
 *==========================================================*/

#ifndef OPSYH
#define OPSYH

#include "conds.h"
#include "token.h"
#include <stdio.h>
#include "node.h"
#include "alfile.h"
#include "errors.h"
#include "symtab.h"
#include "keywords.h"

/* fixed size of array */
#define OP_ARRAY_SIZE 1000

/* type of items in an OP_LIST */
typedef enum{OP_VERB,OP_VAR,OP_STRING,OP_NUM_LIT}OP_TYPE;

/* an OP_LIST is a list of OP_TYPE things which make up
   a COBOL statement */
typedef struct{
              OP_TYPE type;
              union{
                    int keyword_number;
                    NODE *var_name;
                    char *non_numeric_literal;
                    int numeric_literal;
                   }value;
              struct OP_LIST *next;
             }OP_LIST;
```

```
/* an OP_LIST_LIST is a list of statements, these statements
   are contained in the OP_LIST */
typedef struct{
              OP_LIST *op;
              struct OP_LIST_LIST *next;
             }OP_LIST_LIST;

/*FUNCTION DEFS*/

extern OP_LIST_LIST **create_op_array(void);
extern OP_LIST *create_op_list(void);
extern OP_LIST_LIST *create_op_list_list(void);
extern void save_ops_array(char *,OP_LIST_LIST **);
extern void save_op(FILE *,OP_LIST *);
extern void load_ops_array(char *,OP_LIST_LIST **,SYMTAB_ENTRY **);
extern OP_LIST *load_op(FILE *,SYMTAB_ENTRY **,TOKEN *);
extern void display_operations(OP_LIST_LIST **);
extern void display_operation(OP_LIST_LIST *);
extern void evaluate_operation(OP_LIST_LIST *);
extern void evaluate_op(OP_LIST *);
extern void do_move(OP_LIST *);
extern void do_display(OP_LIST *);
extern void do_add(OP_LIST *);


#endif
```

```
/*==============================================================*
 *                    OPS.C                                     *
 * the functions relating to the operations in the interpreter *
 *==============================================================*/


#include "ops.h"

/*==============================================================*
 * create_op_list allocates enough space for an operation list *
 *==============================================================*/
OP_LIST *create_op_list()
    {
        OP_LIST *op=(OP_LIST *)malloc(sizeof(OP_LIST));

        if(op==NULL)
            fatal_error("NO room for OP_LIST");

        op->next=NULL;

        return(op);
    }


/*=========================================================*
 * create_op_list_list allocates enough space for a list *
 * of op_list's                                          *
 *=========================================================*/
OP_LIST_LIST *create_op_list_list()
    {
        OP_LIST_LIST *opl=(OP_LIST_LIST *)malloc(sizeof(OP_LIST_LIST));

        if(opl==NULL)
            fatal_error("NO room for OP_LIST_LIST");

        opl->op=NULL;
        opl->next=NULL;

        return(opl);
```

```
    }

/*==========================================================*
 * the op array is an array of pointers to lists of op_lists *
 * allocate anough space for one fixed size of 999           *
 *==========================================================*/
OP_LIST_LIST **create_op_array()
    {
        int loop;
        OP_LIST_LIST **otp=(OP_LIST_LIST **)malloc(sizeof(OP_LIST_LIST *)
                                                    *OP_ARRAY_SIZE);

        if(otp==NULL)
           fatal_error("no more room for op ARRAY");

        for(loop=0;loop<OP_ARRAY_SIZE;loop++)
           otp[loop]=NULL;

        return(otp);
    }

/*==========================================================*
 * save_ops_array is used because no programs are inverted *
 * loops through array saving ops in the format of         *
 * OP NUM                                                  *
 *   an op                                                 *
 *   any more ops                                          *
 *==========================================================*/
void save_ops_array(char *filename, OP_LIST_LIST **ops_array)
    {
     int loop;
     FILE *outfile=open_output(filename);
     TOKEN token;
     OP_LIST_LIST *opl;
     OP_LIST *op;


     for(loop=0;loop<OP_ARRAY_SIZE;loop++)
        {
```

```
if(ops_array[loop]!=NULL)
  {
    opl=ops_array[loop];

    token.token_type=OP_NUM;
    token.value.keyword_number=loop;
    write_token(outfile,&token);

    while(opl->next!=NULL)
      {
       op=opl->op;
       while(op->next!=NULL)
         {
          save_op(outfile,op);
          op=op->next;
         }
       save_op(outfile,op);

       /* an operation is expected to end
          in a fullstop */

       token.token_type=FULLSTOP;
       token.value.keyword_number=FULLSTOP;
       write_token(outfile,&token);

       opl=opl->next;
      }
      op=opl->op;
      while(op->next!=NULL)
         {
          save_op(outfile,op);
          op=op->next;
         }
      save_op(outfile,op);

      token.token_type=FULLSTOP;
      token.value.keyword_number=FULLSTOP;
      write_token(outfile,&token);
  }
```

```
        }
     close_file(outfile);
}

/*==========================================================*
 * save_op writes out the current part of the op_list_list *
 *==========================================================*/
void save_op(FILE *outfile,OP_LIST *op)
    {

     TOKEN token;

     if(op->type==OP_VERB)
         {
           token.token_type=KEYWORD;
           token.value.keyword_number=op->value.keyword_number;
         }

     if(op->type==OP_VAR)
         {
           token.token_type=VARIABLE_NAME;
           token.value.var_values.unique=
                   op->value.var_name->own.unique;
           token.value.var_values.hash=
                   op->value.var_name->own.hash;
         }

     if(op->type==OP_STRING)
         {
           token.token_type=NON_NUMERIC_LITERAL_LENGTH;
           token.value.keyword_number=
                             strlen(op->value.non_numeric_literal);
           write_token(outfile,&token);
           token.token_type=NON_NUMERIC_LITERAL;
           token.value.string=op->value.non_numeric_literal;
         }

     if(op->type==OP_NUM_LIT)
         {
```

```
            token.token_type=NUMERIC;
            token.value.keyword_number=op->value.numeric_literal;
          }

      write_token(outfile,&token);
    }


/*==============================================================*
 * load_ops_array loads in the operations from filename into *
 * op_array using symtab to find any variables               *
 *==============================================================*/
void load_ops_array(char *filename, OP_LIST_LIST **op_array,
                    SYMTAB_ENTRY **symtab)
    {
     FILE *infile=open_input(filename);
     TOKEN token;
     OP_LIST_LIST *opl;
     OP_LIST *op;
     int eof;


     eof=get_token(infile,&token);  /* get first op_num */

        while(!eof)
            {
             /*create space for op_list_list, stick op in place
               in the array */

             opl=create_op_list_list();
             op_array[token.value.keyword_number]=opl;

             /* get the first part of the first op */
             eof=get_token(infile,&token);

             op=load_op(infile,symtab,&token);
             opl->op=op;

             /* get rest */
```

```
            eof=get_token(infile,&token);

            while(token.token_type!=OP_NUM && !eof)
              {
               opl->next=create_op_list_list();
               op=load_op(infile,symtab,&token);
               opl=opl->next;
               opl->op=op;
               eof=get_token(infile,&token);
              }
            opl->next=NULL; /* do next */

          }
      close_file(infile);
}

/*==================================================================*
 * load_op loads in a full op, and returns it to the op_list_list *
 *==================================================================*/
OP_LIST *load_op(FILE *infile,SYMTAB_ENTRY **symtab,TOKEN *token)
    {

     int eof=0;
     NODE *node;
     OP_LIST *firstop=NULL,*op,*lastop;


           /* load in a full op, which end in a fullstop */
           while(token->token_type!=FULLSTOP && !eof)
              {
               op=create_op_list();

               if(firstop==NULL)
                    firstop=op;
               else
                    lastop->next=op;

               if(token->token_type==KEYWORD)
                  {
```

```
                    op->value.keyword_number=token->value.keyword_number;
                    op->type=OP_VERB;
                   }

                if(token->token_type==VARIABLE_NAME)
                   {
                      op->type=OP_VAR;
                      node=look_for(token->value.var_values.unique,
                                    token->value.var_values.hash,
                                    symtab);
                      op->value.var_name=node;
                   }

                if(token->token_type==NON_NUMERIC_LITERAL)
                   {
                     op->type=OP_STRING;
                     op->value.non_numeric_literal=token->value.string;
                   }

                if(token->token_type==NUMERIC)
                   {
                    op->type=OP_NUM_LIT;
                    op->value.numeric_literal=token->value.keyword_number;
                   }
               lastop=op;
               eof=get_token(infile,token);
              }
      lastop->next=NULL;
      return(firstop);
    }

/*=================================================*
 * display_operations loops through the ops_array *
 * for each op it displays the OP_NUM             *
 * followed by the statements                     *
 *=================================================*/
void display_operations(OP_LIST_LIST **ops_array)
{
   int loop;
```

```
    for(loop=0;loop<OP_ARRAY_SIZE;loop++)
        {
         if(ops_array[loop]!=NULL)
             {
              printf("OP NUM %d\n",loop);
              display_operation(ops_array[loop]);
              printf("\n");
             }
        }
}

/*=================================================*
 * display_operation works through the op_list_list *
 * displaying each op_list on a new line            *
 *=================================================*/
void display_operation(OP_LIST_LIST *op_list_list)
{
    OP_LIST_LIST *opllp;
    OP_LIST *oplp;

    opllp=op_list_list;

    while(opllp!=NULL) /*work through op_list_list */
      {
       oplp=opllp->op;
       while(oplp!=NULL) /* work through op_list */
         {
          if(oplp->type==OP_NUM_LIT) /* display number */
             printf(" %d",oplp->value.numeric_literal);
          else{
              if(oplp->type==OP_STRING) /* display string */
                  printf(" %s",oplp->value.non_numeric_literal);
              else{
                   if(oplp->type==OP_VAR) /* display variable name */
                      printf(" %s",oplp->value.var_name->info->name);
                   else
                      if(oplp->type==OP_VERB) /* display verb number name */
                         printf(" %s",keyword_names[oplp->
```

```
                                                value.keyword_number]);
                    }
                }
            oplp=oplp->next;
          }
        opllp=opllp->next;
        printf("\n");
      }
}

/*=========================================================*
 * evaluate_operation works through a list of statements *
 * and carries each out in turn                          *
 *=========================================================*/
void evaluate_operation(OP_LIST_LIST *op_list_list)
{
    OP_LIST_LIST *opllp;
    OP_LIST *oplp;
    opllp=op_list_list;

    while(opllp!=NULL) /* work through list */
      {
       oplp=opllp->op;
       evaluate_op(oplp); /* evaluate statement */
       opllp=opllp->next;
      }
}

/*=========================================================*
 * evaluate_op determines which type of operation to do *
 *=========================================================*/
void evaluate_op(OP_LIST *op_list)
{
    OP_LIST *oplp;

    oplp=op_list;

    if(oplp->type!=OP_VERB)
       {
```

```
        printf("NOT VERB\n");
        exit();
      }
    else
      {
        switch(oplp->value.keyword_number)
          {
            case MOVE : do_move(oplp); break;
            case ADD  : do_add(oplp); break;
            case DISPLAY : do_display(oplp); break;
            default : printf("verb not implemented\n"); break;
          }
      }
}

void do_move(OP_LIST *op_list)
{

/* valid syntax
**
**   MOVE thing TO variable
**
**   only numeric moves so far
**   between numeric literal and variable
**   and numeric variable and variable    */


    OP_LIST *oplp;
    int value;

    oplp=op_list;        /* skip verb MOVE */
    oplp=oplp->next;

    /* if the 'thing' is a number then use the number
       otherwise use the variables contents */

    if(oplp->type==OP_NUM_LIT)
       value=oplp->value.numeric_literal;
    else
```

```
        value=oplp->value.var_name->info->data_n;

    oplp=oplp->next;     /* skip keyword TO */

    oplp=oplp->next;
    oplp->value.var_name->info->data_n=value; /* allocate value */
                                              /* to variable */
}

void do_display(OP_LIST *op_list)
{
/* valid syntax
**
** DISPLAY thing
** (automatic newline)
** will display the contents of a string or numeric variable
** a string, or a numeric literal
*/
    OP_LIST *oplp;

    oplp=op_list;             /* skip DISPLAY */
    oplp=oplp->next;

    if(oplp->type==OP_STRING)
        printf(" %s\n",oplp->value.non_numeric_literal);
    else
       {
        if(oplp->type==OP_NUM_LIT)
            printf(" %d\n",oplp->value.numeric_literal);
        else
           {
            if(oplp->type==OP_VAR &&
               oplp->value.var_name->info->type==INT)
              printf(" %d\n",oplp->value.var_name->info->data_n);
            else
              printf(" %s\n",oplp->value.var_name->info->data_c);
           }
       }
}
```

```
void do_add(OP_LIST *op_list)
{
/* valid syntax
**
** ADD variable         TO variable
**  or numeric_literal
*/
    int value;
    OP_LIST *oplp=op_list;

    oplp=oplp->next;            /* skip ADD */

    if(oplp->type==OP_NUM_LIT)
       value=oplp->value.numeric_literal;
    else
       value=oplp->value.var_name->info->data_n;

    oplp=oplp->next;       /* skip TO */

    oplp=oplp->next;
    oplp->value.var_name->info->data_n+=value;
}
```

G.1.3 Conditions Table

```
/*==========================================================*
 *                CONDS.H                                   *
 * declare the condition details for the conditions used  *
 * in the interpreter, the different relational operators *
 * used and the limited syntax of the condition           *
 * ie. LEFT_HAND_SIDE RELATION RIGHT_HAND_SIDE            *
 *==========================================================*/

#ifndef CONDHY
#define CONDHY
#include "symtab.h"
#include <stdio.h>
#include "node.h"
#include "alfile.h"
#include "errors.h"
#include "token.h"

#define COND_ARRAY_SIZE 100    /* size of array */


typedef enum{RELATIONAL_NOTHING,
             GREATER_THAN,           /* types of comparison */
             LESS_THAN,              /* possible            */
             EQUAL_TO}REL_OP;

typedef enum{NO_STATUS,YES_STATUS}NOT_STATUS; /* the negation */

typedef enum{SIDE_VAR,                  /* types of things */
             SIDE_NON_LITERAL,          /* that can be     */
             SIDE_LITERAL}SIDE_TYPE;    /* compared        */

/* each side of the condition is of a certain type
   and either points to the contants or in the case
   of a numeric holds it here              */

typedef struct{
               SIDE_TYPE type;
```

```
                union{
                      NODE *variable;
                      char *non_numeric_literal;
                      int numeric_literal;
                     }contents;
               }CONDITION_SIDE;

/* the relation between the two sides has a negation status
   and the type of comparison to take place                 */

typedef struct{
               NOT_STATUS not;
               REL_OP relation;
              }RELATIONAL;

/* the actual, lhs comparison rhs , condition */
typedef struct{
               CONDITION_SIDE lhs;
               RELATIONAL relation;
               CONDITION_SIDE rhs;
              }COND;

/*FUNCTION DEFS*/

extern COND **create_cond_array(void);
extern COND *create_cond(void);
extern void save_cond_array(char *,COND **);
extern void save_side(CONDITION_SIDE *,FILE *);
extern void load_cond_array(char *,COND **,SYMTAB_ENTRY **);
extern void load_side(CONDITION_SIDE *,TOKEN *,SYMTAB_ENTRY **);
extern void display_condition(COND *);
extern void display_side(CONDITION_SIDE *);
extern int evaluate_condition(COND *);


#endif
```

```
/*=========================================================*
 *                    CONDS.C                              *
 * The functions used to manipulate the condition          *
 * representations used by the interpreter.                *
 * The save and load functions were written out of         *
 * necessity because no inversion was used to link the     *
 * main interpreter modules                                *
 *=========================================================*/

#include "conds.h"

/*==========================================================*
 * create_cond allocates memory for a condition and returns *
 * the address                                              *
 *==========================================================*/
COND *create_cond()
    {
        COND *cond=(COND *)malloc(sizeof(COND));

        if(cond==NULL)
            fatal_error("NO room for COND");

        return(cond);
    }


/*==========================================================*
 * create_cond_array creates anough memory for an array of  *
 * pointers to conditions, the array is of fixed size       *
 * 99 because this is the number of conditions available    *
 * in probench                                              *
 *==========================================================*/
COND **create_cond_array()
    {
        int loop;
        COND **ctp=(COND **)malloc(sizeof(COND *)*COND_ARRAY_SIZE);

        if(ctp==NULL)
           fatal_error("no more room for cond ARRAY");
```

```
        for(loop=0;loop<COND_ARRAY_SIZE;loop++) /*initialise array */
           ctp[loop]=NULL;

        return(ctp);
    }

/*==============================================================*
 * save_cond_array opens filename and loops through the array *
 * writing out the condition number ( the loop position )     *
 * writing out the left hand side details,                    *
 * then the relational operator                               *
 * and then the right hand side                               *
 *==============================================================*/
void save_cond_array(char *filename, COND **cond_array)
    {
     int loop;
     FILE *outfile=open_output(filename);
     TOKEN token;


     for(loop=0;loop<COND_ARRAY_SIZE;loop++)
        {
         if(cond_array[loop]!=NULL)
           {
             token.token_type=COND_NUM;        /* save cond num */
             token.value.keyword_number=loop;
             write_token(outfile,&token);

             /* save left hand side */
             save_side(&(cond_array[loop]->lhs),outfile);

             /* the relational op consists of two parts
                the not status followed by the relation */

             token.token_type=RELATIONAL_OP;
             token.value.keyword_number=cond_array[loop]->
                                          relation.not;
             write_token(outfile,&token);
```

```
            token.token_type=RELATIONAL_OP;
            token.value.keyword_number=cond_array[loop]->
                                        relation.relation;
            write_token(outfile,&token);

            /* save right hand side */
            save_side(&(cond_array[loop]->rhs),outfile);
          }
        }
     close_file(outfile);
    }

/*==============================================================*
 * save_side determines which type of token to write out       *
 * dependant upon what is stored at that side of the condition *
 *==============================================================*/
void save_side(CONDITION_SIDE *cond_side,FILE *outfile)
    {
     TOKEN token;
     if(cond_side->type==SIDE_VAR)
        {
         token.token_type=VARIABLE_NAME;
         token.value.var_values.unique=
                 cond_side->contents.variable->own.unique;
         token.value.var_values.hash=
                 cond_side->contents.variable->own.hash;
        }
     if(cond_side->type==SIDE_LITERAL)
        {
         token.token_type=NUMERIC;
         token.value.keyword_number=cond_side->contents.
                                   numeric_literal;
        }
     if(cond_side->type==SIDE_NON_LITERAL)
        {

         token.token_type=NON_NUMERIC_LITERAL_LENGTH;
         token.value.keyword_number=strlen(cond_side->contents.
                                          non_numeric_literal);
```

```
        write_token(outfile,&token);
        token.token_type=NON_NUMERIC_LITERAL;
        token.value.string=cond_side->contents.non_numeric_literal;
       }
        write_token(outfile,&token);
    }

/*==============================================================*
 * load_cond_array uses the file created by the above code to *
 * build up a condition array, the symbol table is used to    *
 * find the addresses of the variables in the conditions      *
 *==============================================================*/
void load_cond_array(char *filename, COND **cond_array,
                     SYMTAB_ENTRY **symtab)
    {
     int eof;
     FILE *infile=open_input(filename);
     TOKEN token;
     COND *condition;

     eof=get_token(infile,&token); /* get the first condition number */

     while(!eof)
      {
      condition=create_cond();                          /* put the new  */
      cond_array[token.value.keyword_number]=condition; /* condition in */
                                                        /*the slot     */

      eof=get_token(infile,&token);                     /* get side        */
      load_side(&(condition->lhs),&token,symtab);       /* place it on cond */

      eof=get_token(infile,&token);                      /* get relation */
      condition->relation.not=token.value.keyword_number;/* not status   */

      eof=get_token(infile,&token);                      /* get the relation *
      condition->relation.relation=token.value.keyword_number;

      eof=get_token(infile,&token);                 /* get the side */
      load_side(&(condition->rhs),&token,symtab);   /* place it on cond */
```

```
      eof=get_token(infile,&token); /* get any more */
      }

}

/*==============================================================*
 * load_side takes a previously loaded token and puts it in the *
 * condition correctly                                          *
 *==============================================================*/
void load_side(CONDITION_SIDE *side,TOKEN *token,
               SYMTAB_ENTRY **symtab)
    {
       if(token->token_type==VARIABLE_NAME)
           {
             /* find the variable using the hash value
                and the unique number stored on file */
             side->type=SIDE_VAR;
             side->contents.variable=look_for(
                                     token->value.var_values.unique,
                                     token->value.var_values.hash,
                                     symtab);
           }

       if(token->token_type==NON_NUMERIC_LITERAL)
          {
             /* use the string loaded in by the token code */
             side->type=SIDE_NON_LITERAL;
             side->contents.non_numeric_literal=token->value.string;
          }

      if(token->token_type==NUMERIC)
          {
           /* allocate the numeric value */
           side->type=SIDE_LITERAL;
           side->contents.numeric_literal=token->value.keyword_number;
          }

}
```

```
/*==============================================================*
 * display_conditions loops through the array                   *
 * displaying the condition number, followed by the condition *
 * itself                                                       *
 *==============================================================*/
void display_conditions(COND **cond_array)
{
    int loop;

    for(loop=0;loop<COND_ARRAY_SIZE;loop++)
        {
         if(cond_array[loop]!=NULL)
           {
            printf("COND %d\n",loop);
            display_condition(cond_array[loop]);
            printf("\n");
           }
        }
}

/*==============================================*
 * display_condition takes a condition,         *
 * displays the left hand side,                 *
 * if the not status is set then displays NOT   *
 * then the relational operator                 *
 * followed by the right hand side              *
 *==============================================*/
void display_condition(COND *cond)
{
    display_side(&(cond->lhs));

    if(cond->relation.not)
         printf(" NOT ");

    if(cond->relation.relation==GREATER_THAN)
         printf(" >");
    else{
         if(cond->relation.relation==LESS_THAN)
```

```
                printf(" <");
            else
                if(cond->relation.relation==EQUAL_TO)
                    printf(" =");
            }

    display_side(&(cond->rhs));
}

/*=====================================================*
 * display_side displays the relevant details of the *
 * conditional side,                                 *
 * for variables it displays the name                *
 * strings display their contents                    *
 * and numbers get the number displayed              *
 *=====================================================*/
void display_side(CONDITION_SIDE *side)
{
    if(side->type==SIDE_VAR)
        printf(" %s",side->contents.variable->info->name);
    else{
         if(side->type==SIDE_NON_LITERAL)
              printf(" %s",side->contents.non_numeric_literal);
         else
             if(side->type==SIDE_LITERAL)
                 printf(" %d",side->contents.numeric_literal);
        }
}

/*=================================================================*
 * evaluate condition takes a condition, and returns the          *
 * boolean value that it evaluates to.                            *
 *                                                                *
 * Currently equality comparisons are allowed between             *
 * strings and string variables, strings and strings, and         *
 * string variables and string variables                          *
 *                                                                *
 * Numeric comparisons can take place between numeric literals    *
 * and numeric variables in any combination as above.             *
```

```
 * The comparisons available are >, <, =, NOT <, NOT >, and NOT = *
 *                                                                 *
 * The type of condition is determined by the left hand side so    *
 * the conditions must be semantically correct with no mixed       *
 * conditions i.e numeric < string etc.                            *
 *=================================================================*/
int evaluate_condition(COND *cond)
{
    int lhs_num,rhs_num;
    char *lhs,*rhs;
    int truth=0;            /* ever the optomist */

    /*if string or string variable on left hand side*/
    if(cond->lhs.type==SIDE_NON_LITERAL ||
       (cond->lhs.type == SIDE_VAR &&
        cond->lhs.contents.variable->info->type==CHAR))
      {
       if(cond->relation.relation==EQUAL_TO) /* must be == */
         {

          /* if the side is a string then point to the
             string, if it is a variabel the point to
             the variables contents                    */

          if(cond->lhs.type==SIDE_NON_LITERAL)
             lhs=cond->lhs.contents.non_numeric_literal;
          else
             lhs=cond->lhs.contents.variable->info->data_c;

          if(cond->rhs.type==SIDE_NON_LITERAL)
             rhs=cond->rhs.contents.non_numeric_literal;
          else
             rhs=cond->rhs.contents.variable->info->data_c;

          truth=!strcmp(lhs,rhs); /* compare them */
         }
      }
    else
       {
```

```
        /* if the side is a number then get the number
           if it is a variable then get the variable contents */

        if(cond->rhs.type==SIDE_LITERAL)
            rhs_num=cond->rhs.contents.numeric_literal;
        else
            rhs_num=cond->rhs.contents.variable->info->data_n;

        if(cond->lhs.type==SIDE_LITERAL)
            lhs_num=cond->lhs.contents.numeric_literal;
        else
            lhs_num=cond->lhs.contents.variable->info->data_n;

        /* carry out the comparison and determine the truth value */

        if(cond->relation.relation==EQUAL_TO)
              truth=(lhs_num==rhs_num) ? 1:0;
        else{
             if(cond->relation.relation==GREATER_THAN)
                  truth=(lhs_num>rhs_num) ? 1:0;
             else
                if(cond->relation.relation==LESS_THAN)
                   truth=(lhs_num<rhs_num) ? 1:0;

            }
       }
    /* if the NOT status is set then negate the truth value
       otherwise return the truth value as is */

    return((cond->relation.not) ? !truth:truth);
}
```

G.1.4 Stack By Array

```
/*==================================================*
 *              STCKBYAR.H                          *
 * the functions for a stack by array in c based on *
 * formal spec presented by K. Barclay              *
 *==================================================*/
#ifndef STCKBYAR
#define STCKBYAR

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

typedef struct{
                NODE **array;
                int stack_max_size;
                int stack_curr_size;
              }STACKBYARRAY;

/* POP REMOVES THE TOP OF STACK */

extern STACKBYARRAY *stackpop(STACKBYARRAY *);
extern NODE *stacktop(STACKBYARRAY *);
extern STACKBYARRAY *stackpush(STACKBYARRAY *,NODE *);
extern STACKBYARRAY *create_stack_by_array(int);

#endif
```

```
/*=====================================*
 *                STCKBYAR.C           *
 * functions for stack by array in C   *
 *=====================================*/

#include "stckbyar.h"
/* POP REMOVES THE TOP OF STACK */

STACKBYARRAY *stackpop(STACKBYARRAY *pstck)
     {
      if(pstck->stack_curr_size==0)
            pstck->array[0]=NULL;
      else
        {
         pstck->array[pstck->stack_curr_size--]=NULL;
        }
      return pstck;
     }

/* return top of stack */
NODE *stacktop(STACKBYARRAY *pstck)
     {
      if(pstck->stack_curr_size==0)
          return NULL;
      else
         return(pstck->array[pstck->stack_curr_size-1]);
      }

/* place an item on the stack */
STACKBYARRAY *stackpush(STACKBYARRAY *pstck,NODE *pnode)
      {
       if( pstck->stack_curr_size!=pstck->stack_max_size)
           pstck->array[pstck->stack_curr_size++]=pnode;
       return pstck;
      }

/* create the array dynamically */
STACKBYARRAY *create_stack_by_array(int size)
   {
```

```
    STACKBYARRAY *sbap=(STACKBYARRAY *)malloc(sizeof(STACKBYARRAY));

    if(sbap==NULL)
          fatal_error("no more room for stack by array");

    sbap->array=(NODE **)malloc(sizeof(NODE *)*size);

    if(sbap->array==NULL)
         fatal_error("no more room for ARRAY in Stack by array");

    sbap->stack_max_size=size;
    sbap->stack_curr_size=0;

    return(sbap);
}
```

G.1.5 Trie

```
/*=====================================================*
 *            STAGLIST.H                               *
 * defines the functions necessary to maintain a trie *
 *=====================================================*/

#ifndef STAGGERED_LIST
#define STAGGERED_LIST

#include <stdio.h>
#include "errors.h"
#include "alfile.h"


#define VALID_CHARS     27


struct stag_list_struct{
                        struct stag_list_struct *pointer[VALID_CHARS];
                        int data;
                  };

typedef struct stag_list_struct staggered_list;

extern staggered_list *create_list();
extern int get_data(staggered_list *);
extern void put_data(staggered_list *,int);
extern staggered_list *add_pointer(staggered_list *,char);
extern staggered_list *check_pointer(staggered_list *,char);
extern staggered_list *create_list_with_file(char *);
#endif
```

```
/*==============================================*
 *        STAGLIST.C                            *
 * an implementation of a trie for keyword names *
 * memory intensive and does not work on a PC    *
 *==============================================*/

#include "staglist.h"

#define DATA_FIELD              27
#define HYPHEN_FIELD    26

/*============================*
 * allocates memory for list *
 *============================*/
staggered_list *create_list()
        {
                int loop;
                staggered_list *stag;
                stag= (staggered_list *)malloc(sizeof(staggered_list));
                if(stag==NULL)
                 fatal_error("not enough memory for staggered list segment");

                for(loop=0;loop<27;loop++)
                        stag->pointer[loop]=NULL;
                stag->data=NULL;

                return stag;
        }

/*=========================================*
 * return the data held at this trie node *
 *=========================================*/
int get_data(staggered_list *stag)
        {
                return stag->data;
        }

/*=====================================*
 * insert data at this trie node       *
```

```
 *=====================================*/
void put_data(staggered_list *stag,int put)
        {
                stag->data=put;
        }

/*==============================================================*
 * work out what the offset in the list for this character     *
 * is and if it is null then create a new list segment for it *
 * then return the address of the segment                      *
 *==============================================================*/
staggered_list *add_pointer(staggered_list *stag,char ch)
     {
       int offset=(ch=='-') ? HYPHEN_FIELD : (ch-'A');

       if(stag->pointer[offset]==NULL)
          stag->pointer[offset]=create_list();

       return(stag->pointer[offset]);

     }
/*==============================================================*
 * retrieves the pointer for a stag list segment at a character *
 * offset                                                       *
 *==============================================================*/
staggered_list *check_pointer(staggered_list *stag,char ch)
        {
            return(stag->pointer[(ch=='-') ? HYPHEN_FIELD : (ch-'A')]);
        }

/*==============================================================*
 * builds up a trie using data from the file, the unique data *
 * number is only incremented when a field with an * is found *
 * this allows verbs such as ZERO,ZEROES to share the same     *
 * keyword value, since they are effectively the same keyword *
 *==============================================================*/
staggered_list *create_list_with_file(char *infile_name)
  {
     FILE *infile=open_input(infile_name);
```

```
    char ch;
    staggered_list *list_pointer,*root=create_list();
    int data_count=0;

    ch=getc(infile);

    data_count= (ch=='*') ? 0 : 1;  /* 0 reserved for undefined etc*/

    while(ch!=EOF)
       {
          if(ch=='*'){
                      data_count++;
                      ch=getc(infile);
                      }

          list_pointer=root;

          while(ch!='\n' && ch!=EOF)
               {

                 list_pointer=add_pointer(list_pointer,ch);
                 ch=getc(infile);
               }
          if(ch==EOF)
              fatal_error("unexpected end of file in list setup");

          put_data(list_pointer,data_count);
          ch=getc(infile);
         }
    close_file(infile);
    return root;
  }
```

G.1.6 Intermediate File Tokens

```
/*==============================================================*
 *                    TOKEN.H                                   *
 * defines the token functions and types, used for intermediate *
 * file access                                                  *
 *==============================================================*/
#ifndef TOKENYH
#define TOKENYH

#include <stdio.h>
#include "node.h"
#include "symtab.h"

typedef enum{NOTHING,OPEN_BRACE,CLOSE_BRACE,FULLSTOP,
             KEYWORD,NON_KEYWORD,NON_KEYWORD_LENGTH,
             NON_NUMERIC_LITERAL,NON_NUMERIC_LITERAL_LENGTH,
             OP_NUM,VARIABLE_NAME,NUMERIC,COND_NUM,RELATIONAL_OP
            }TOKEN_TYPE;

typedef struct{
               TOKEN_TYPE token_type;
               union{
                     int keyword_number;
                     char *string;
                     save_values var_values;
                    }value;
              }TOKEN;

extern void write_token(FILE *,TOKEN *);
extern int get_token(FILE *,TOKEN *);

#endif
```

```
/*==============================================================*
 *                          TOKEN.C                             *
 * the two token functions write and get for intermediate file *
 * access                                                       *
 *==============================================================*/

#include "token.h"

/*==============================================================*
 * write token writes out a token, this is done in several ways *
 * dependant upon the token type                                *
 *==============================================================*/
void write_token(FILE *fp,TOKEN *toke)
{
    fprintf(fp,"%d\n",toke->token_type); /* write out token type */

    if(toke->token_type==OPEN_BRACE ||     /* if (,), or . then  */
       toke->token_type==CLOSE_BRACE ||    /* write out the type */
       toke->token_type==FULLSTOP){        /* again              */
         fprintf(fp,"%d\n",toke->token_type);
       }
    else{
         if(toke->token_type==KEYWORD ||
            toke->token_type==OP_NUM ||
            toke->token_type==NUMERIC ||
            toke->token_type==COND_NUM ||
            toke->token_type==RELATIONAL_OP ||
            toke->token_type==NON_KEYWORD_LENGTH ||
            toke->token_type==NON_NUMERIC_LITERAL_LENGTH){
              fprintf(fp,"%d\n",toke->value.keyword_number);
            }
         else{   /* if its a string type then write it out */
              if(toke->token_type==NON_NUMERIC_LITERAL ||
                 toke->token_type==NON_KEYWORD){
                   fprintf(fp,"%s\n",toke->value.string);
                }
              else{  /* if its a variable write out hash then unique */
                   if(toke->token_type==VARIABLE_NAME){
                      fprintf(fp,"%d\n",toke->value.var_values.hash);
```

```
                    fprintf(fp,"%d\n",toke->value.var_values.unique);
                  }
                }
            }
        }
}


/*==============================================================*
 * get_token reads a token from the file, first get the token *
 * type, then read the relevant details                        *
 *==============================================================*/
int get_token(FILE *fp,TOKEN *toke)
 {

char g,*str;
int size;

fscanf(fp,"%d",&(toke->token_type)); /* read token type */

if(feof(fp)) return(1);

     g=getc(fp);              /* read \n */
     if(g!='\n')
          {
           printf("token number READ ERROR\n");
           exit(0);
          }


if(toke->token_type==OPEN_BRACE ||        /* get type again */
   toke->token_type==CLOSE_BRACE ||       /* then get \n    */
   toke->token_type==FULLSTOP)
     {
     fscanf(fp,"%d",&(toke->value.keyword_number));
     g=getc(fp);
     if(g!='\n')
          {
```

```
            printf("open brace etc READ ERROR\n");
            exit(0);
            }
        }
else{
        if(toke->token_type==KEYWORD ||       /* if one of these */
           toke->token_type==OP_NUM ||        /* then read the   */
           toke->token_type==NUMERIC ||       /* integer value   */
           toke->token_type==COND_NUM ||
           toke->token_type==RELATIONAL_OP ){
                fscanf(fp,"%d",&(toke->value.keyword_number));
                g=getc(fp);
                if(g!='\n')        /* get \n */
                 {
                  printf("keyword num etc READ ERROR\n");
                  exit(0);
                }
            }
        else{
            if(toke->token_type==VARIABLE_NAME) /* if variable */
              {
                /* get hash value */
                fscanf(fp,"%d",&(toke->value.var_values.hash));
                g=getc(fp);
                if(g!='\n')
                  {
                   printf("hash num etc READ ERROR\n");
                   exit(0);
                  }

                /* get unique value */
                fscanf(fp,"%d",&(toke->value.var_values.unique));
                g=getc(fp);
                if(g!='\n')
                  {
                   printf("unique num etc READ ERROR\n");
                   exit(0);
                  }
              }
```

```
        else          /* else it must be a string */
         {
          fscanf(fp,"%d",&size);   /* read the size */
               g=getc(fp);
               if(g!='\n')
               {
               printf("size READ ERROR\n");
               exit(0);
               }
            /* allocate the size */
          toke->value.string=(char *)malloc(size+1);
          str=toke->value.string;

              /* read the string , type first*/
              fscanf(fp,"%d",&(toke->token_type));
               g=getc(fp);
               if(g!='\n')
                {
                 printf("type READ ERROR\n");
                 exit(0);
                }

          /* then read the string */
          g=getc(fp);
          while(g!='\n'){
               *str=g;
                str++;
               *str='\0';
               g=getc(fp);
               }
          }
       }
    }
return(0);
}
```

G.1.7 Keyword Definitions

```
/*=====================================================*
 *                    KEYWORDS.H                       *
 * enumerated type of all the reserved words in COBOL  *
 * plus a global array of text strings for the variable *
 * names                                               *
 *=====================================================*/

#ifndef RESERVED_WORDS
#define RESERVED_WORDS

typedef enum{DUMMY,
ACCEPT,ACCESS,ADD,ADVANCING,
AFTER,ALL,ALPHABET,ALPHABETIC,ALPHABETIC_LOWER,ALPHABETIC_UPPER,
ALPHANUMERIC,ALPHANUMERIC_EDITED,ALSO,ALTER,
ALTERNATE,AND,ANY,ARE,AREA,ASCENDING,ASSIGN,AT,AUTHOR,
BEFORE,BINARY,BLANK,BLOCK,BOTTOM,BY,CALL,CANCEL,CD,CF,
CH,CHARACTER,CHARACTERS,CLASS,CLOSE,
CODE,CODE_SET,COLLATING,COLUMN,COMMA,
COMMON,COMMUNICATION,COMP,COMPUTE,CONFIGURATION,
CONTAINS,CONTENT,CONTINUE,CONTROL,CONTROLS,
CONVERTING,COPY,CORRESPONDING,COUNT,CURRENCY,
DATA,DATE,DATE_COMPILED,DATE_WRITTEN,DAY,
DAY_OF_WEEK,DE,DEBUG_CONTENTS,DEBUG_ITEM,DEBUG_LINE,
DEBUG_NAME,DEBUG_SUB_1,DEBUG_SUB_2,DEBUG_SUB_3,DEBUGGING,
DECIMAL_POINT,DECLARATIVES,DELETE,DELIMITED,DELIMITER,
DEPENDING,DESCENDING,DESTINATION,DETAIL,DISABLE,
DISPLAY,DIVIDE,DIVISION,DOWN,DUPLICATES,DYNAMIC,
EGI,ELSE,EMI,ENABLE,END,END_ADD,
END_CALL,END_COMPUTE,END_DELETE,END_DIVIDE,END_EVALUATE,
END_IF,END_MULTIPLY,END_OF_PAGE,END_PERFORM,END_READ,
END_RECEIVE,END_RETURN,END_REWRITE,END_SEARCH,END_START,END_STRING,
END_SUBTRACT,END_UNSTRING,END_WRITE,ENVIRONMENT,EOP,
EQUAL,ERROR,ESI,EVALUATE,EXCEPTION,EXIT,
EXTEND,EXTERNAL,FALSE,FD,COB_VERB_FILE,FILE_CONTROL,
FILLER,FINAL,FIRST,FOOTING,FOR,FROM,
GENERATE,GIVING,GLOBAL,GO,GREATER,GROUP,
HEADING,HIGH_VALUES,I_O,I_O_CONTROL,IDENTIFICATION,IF,
```

```
IN,INDEX,INDEXED,INDICATE,INITIAL,INITIALIZE,
INITIATE,INPUT,INPUT_OUTPUT,INSPECT,INSTALLATION,INTO,
INVALID,IS,JUST,JUSTIFIED,KEY,LABEL,
LAST,LEADING,LEFT,LENGTH,LESS,LIMIT,
LIMITS,LINAGE,LINAGE_COUNTER,LINE,LINE_COUNTER,LINES,
LINKAGE,LOCK,LOW_VALUES,MERGE,MESSAGE,MODE,
MOVE,MULTIPLE,MULTIPLY,NATIVE,NEGATIVE,NEXT,
NO,NOT,NUMBER,COB_VERB_NUMERIC,NUMERIC_EDITED,OBJECT_COMPUTER,
OCCURS,OF,OFF,OMITTED,ON,OPEN,
OPTIONAL,OR,ORDER,ORGANIZATION,OTHER,OUTPUT,
OVERFLOW,PACKED_DECIMAL,PADDING,PAGE,PAGE_COUNTER,PERFORM,
PF,PH,PIC,PLUS,POINTER,POSITION,
POSITIVE,PRINTING,PROCEDURE,PROCEDURES,PROCEED,PROGRAM,
PROGRAM_ID,PURGE,QUEUE,QUOTES,RANDOM,RD,READ,RECEIVE,RECORDS,
REDEFINES,REEL,REFERENCES,RELATIVE,RELEASE,REMAINDER,
EMOVAL,RENAMES,REPLACE,REPLACING,REPORT,REPORTING,REPORTS,
RESERVE,RESET,RETURN,REWIND,REWRITE,RF,RH,RIGHT,
COB_VERB_ROUNDED,RUN,SAME,SD,
SEARCH,SECTION,SECURITY,SEGMENT,SEGMENT_LIMIT,SELECT,SEND,SENTENCE,
SEPARATE,COB_VERB_SEQUENCE,SEQUENTIAL,SET,SIGN,SIZE,SORT,SORT_MERGE,SOURCE,
SOURCE_COMPUTER,SPACES,SPECIAL_NAMES,STANDARD,STANDARD_1,STANDARD_2,START,
STATUS,STOP,COB_VERB_STRING,SUB_QUEUE_1,SUB_QUEUE_2,SUB_QUEUE_3,SUBTRACT,
SUM,SUPPRESS,SYMBOLIC,SYNCHRONIZED,TABLE,TALLYING,TAPE,TERMINAL,
TERMINATE,TEST,TEXT,THAN,THEN,THRU,TIME,TIMES,TO,TOP,TRAILING,
TRUE,COB_VERB_TYPE,UNIT,UNSTRING,UNTIL,UP,UPON,USAGE,USE,USING,VALUES,
VARYING,WHEN,WITH,WORKING_STORAGE,WRITE,ZERO
}RESERVED_WORD;

char *keyword_names[]={"DUMMY",
"ACCEPT","ACCESS","ADD","ADVANCING",
"AFTER","ALL","ALPHABET","ALPHABETIC","ALPHABETIC_LOWER","ALPHABETIC_UPPER",
"ALPHANUMERIC","ALPHANUMERIC_EDITED","ALSO","ALTER",
"ALTERNATE","AND","ANY","ARE","AREA","ASCENDING","ASSIGN","AT","AUTHOR",
"BEFORE","BINARY","BLANK","BLOCK","BOTTOM","BY","CALL","CANCEL","CD","CF",
"CH","CHARACTER","CHARACTERS","CLASS","CLOSE",
"CODE","CODE_SET","COLLATING","COLUMN","COMMA",
"COMMON","COMMUNICATION","COMP","COMPUTE","CONFIGURATION",
"CONTAINS","CONTENT","CONTINUE","CONTROL","CONTROLS",
"CONVERTING","COPY","CORRESPONDING","COUNT","CURRENCY",
```

```
"DATA","DATE","DATE_COMPILED","DATE_WRITTEN","DAY",
"DAY_OF_WEEK","DE","DEBUG_CONTENTS","DEBUG_ITEM","DEBUG_LINE",
"DEBUG_NAME","DEBUG_SUB_1","DEBUG_SUB_2","DEBUG_SUB_3","DEBUGGING",
"DECIMAL_POINT","DECLARATIVES","DELETE","DELIMITED","DELIMITER",
"DEPENDING","DESCENDING","DESTINATION","DETAIL","DISABLE",
"DISPLAY","DIVIDE","DIVISION","DOWN","DUPLICATES","DYNAMIC",
"EGI","ELSE","EMI","ENABLE","END","END_ADD",
"END_CALL","END_COMPUTE","END_DELETE","END_DIVIDE","END_EVALUATE",
"END_IF","END_MULTIPLY","END_OF_PAGE","END_PERFORM","END_READ",
"END_RECEIVE","END_RETURN","END_REWRITE","END_SEARCH","END_START","END_STRING'
"END_SUBTRACT","END_UNSTRING","END_WRITE","ENVIRONMENT","EOP",
"EQUAL","ERROR","ESI","EVALUATE","EXCEPTION","EXIT",
"EXTEND","EXTERNAL","FALSE","FD","FILE","FILE_CONTROL",
"FILLER","FINAL","FIRST","FOOTING","FOR","FROM",
"GENERATE","GIVING","GLOBAL","GO","GREATER","GROUP",
"HEADING","HIGH_VALUES","I_O","I_O_CONTROL","IDENTIFICATION","IF",
"IN","INDEX","INDEXED","INDICATE","INITIAL","INITIALIZE",
"INITIATE","INPUT","INPUT_OUTPUT","INSPECT","INSTALLATION","INTO",
"INVALID","IS","JUST","JUSTIFIED","KEY","LABEL",
"LAST","LEADING","LEFT","LENGTH","LESS","LIMIT",
"LIMITS","LINAGE","LINAGE_COUNTER","LINE","LINE_COUNTER","LINES",
"LINKAGE","LOCK","LOW_VALUES","MERGE","MESSAGE","MODE",
"MOVE","MULTIPLE","MULTIPLY","NATIVE","NEGATIVE","NEXT",
"NO","NOT","NUMBER","NUMERIC","NUMERIC_EDITED","OBJECT_COMPUTER",
"OCCURS","OF","OFF","OMITTED","ON","OPEN",
"OPTIONAL","OR","ORDER","ORGANIZATION","OTHER","OUTPUT",
"OVERFLOW","PACKED_DECIMAL","PADDING","PAGE","PAGE_COUNTER","PERFORM",
"PF","PH","PIC","PLUS","POINTER","POSITION",
"POSITIVE","PRINTING","PROCEDURE","PROCEDURES","PROCEED","PROGRAM",
"PROGRAM_ID","PURGE","QUEUE","QUOTES","RANDOM","RD","READ","RECEIVE","RECORDS'
"REDEFINES","REEL","REFERENCES","RELATIVE","RELEASE","REMAINDER",
"REMOVAL","RENAMES","REPLACE","REPLACING","REPORT","REPORTING","REPORTS",
"RESERVE","RESET","RETURN","REWIND","REWRITE","RF","RH","RIGHT","ROUNDED",
"RUN","SAME","SD",
"SEARCH","SECTION","SECURITY","SEGMENT","SEGMENT_LIMIT","SELECT",
"SEND","SENTENCE",
"SEPARATE","SEQUENCE","SEQUENTIAL","SET","SIGN","SIZE",
"SORT","SORT_MERGE","SOURCE",
"SOURCE_COMPUTER","SPACES","SPECIAL_NAMES","STANDARD",
```

```
"STANDARD_1","STANDARD_2","START",
"STATUS","STOP","STRING","SUB_QUEUE_1",
"SUB_QUEUE_2","SUB_QUEUE_3","SUBTRACT",
"SUM","SUPPRESS","SYMBOLIC","SYNCHRONIZED",
"TABLE","TALLYING","TAPE","TERMINAL",
"TERMINATE","TEST","TEXT","THAN","THEN","THRU",
"TIME","TIMES","TO","TOP","TRAILING",
"TRUE","TYPE","UNIT","UNSTRING","UNTIL","UP",
"UPON","USAGE","USE","USING","VALUES",
"VARYING","WHEN","WITH","WORKING_STORAGE","WRITE","ZERO"
};

#endif
```

G.1.8 Trace Functions

The following code uses the interpreter information and also information from the diagrammer.

```
/*=========================================================*
 *                        TRACE.H                          *
 * define the trace structure and the debugger functions *
 *=========================================================*/

#ifndef TRACEYH
#define TRACEYH

#include <stdio.h>
#include "ops.h"
#include "conds.h"
#include "alpro.h"

/* a run consists of the current box, the last box number visited *
 * the current op, the op number position in the list, the current*
 * tree, current symbol table, current op array, and the finished *
 * indicator                                                      */

struct run_info{
                BOX *current_box;
                int last_box_num;
                OPLIST *current_op;
                int op_num;
                TREE *current_tree;
                SYMTAB_ENTRY **current_symbols;
                COND **current_conds;
                OP_LIST_LIST **current_ops;
                int finished;
              };

typedef struct run_info RUN_INFO;
```

```
extern RUN_INFO *create_run(TREE *,SYMTAB_ENTRY **,COND **,OP_LIST_LIST **);
extern void reset_run(RUN_INFO *);
extern int get_next_op(RUN_INFO *);
extern int next_op(RUN_INFO *);
extern int next_box_from(RUN_INFO *);
extern int get_next_op(RUN_INFO *);

#endif
```

```
/*=============================================*
 *                        TRACE.C              *
 * all the trace functions                     *
 *=============================================*/

#include "trace.h"

/*==============================================================*
 * create_run allocates space for run details, and initialise *
 * the values                                                 *
 *==============================================================*/
RUN_INFO *create_run(TREE *tree,SYMTAB_ENTRY **symtab,
                     COND **cond_array,OP_LIST_LIST **ops_array)
{
    RUN_INFO *run;

        if((run=(RUN_INFO *)malloc(sizeof(RUN_INFO)))==NULL)
                fatal_error("no more room for run info");


    run->current_box=tree->root;
    run->last_box_num=0;
    run->current_op=tree->root->op;
    run->op_num=0;
    run->current_tree=tree;
    run->current_symbols=symtab;
    run->current_conds=cond_array;
    run->current_ops=ops_array;
    run->finished=0;

    return(run);
}

/*==============================================================*
 * reset_run returns all the run variables to their original *
 *==============================================================*/
void reset_run(RUN_INFO *run)
{
    run->current_box=run->current_tree->root;
```

```
    run->last_box_num=0;
    run->current_op=run->current_tree->root->op;
    run->op_num=0;
    reset_symtab(run->current_symbols,100);
    run->finished=0;
}

/*==================================================================*
 * get_next_op uses the current run details to determine whether *
 * another operation is available, and if so alters the run      *
 * details                                                        *
 *==================================================================*/
int get_next_op(RUN_INFO *info)
{
    BOXLIST *blp;

    /* if the current box has ops then check the box
       to see if there are any ops left to do */
    if(info->current_box->op!=NULL && next_op(info))
       {
        /* make this box the last visited box */
        info->last_box_num=info->current_box->box_number;
        return(1);
       }
    else
       {           /* box has kids*/
       if(info->current_box->children!=NULL)
         {

         /* if the current box is a selection then work
            through the children testing the conditions
            and if any pass then make that box the current
            and check it for ops */

         if(info->current_box->probench_box.box_type==SELECTION)
           {
            blp=info->current_box->children;
            while(blp!=NULL)
             {
```

```
        if(evaluate_condition(info->current_conds[
                                blp->box->probench_box.cond_num]))
         {
          info->current_box=blp->box;
          info->last_box_num=blp->box->box_number;
          return(get_next_op(info));
         }
        blp=blp->link;
       }
     /* if none then work back to find the next box and
        check it for ops */
     if(next_box_from(info))
      return(get_next_op(info));
     else
       return(0);
     }

     /* if the box is an iteration then check the child's
        condition, if the negation of it succeeds then
        make the child the current box */
   if(info->current_box->probench_box.box_type==ITERATION)
     {
      blp=info->current_box->children;
        if(!evaluate_condition(info->current_conds[
                                blp->box->probench_box.cond_num]))
         {
          info->current_box=blp->box;
          info->last_box_num=blp->box->box_number;
          return(get_next_op(info));
         }
     /* if none then work back to find the next box and
        check it for ops */
     if(next_box_from(info))
      return(get_next_op(info));
     else
       return(0);
    }

   /* if the box is a sequence then make the first child
```

```
            the current box, and check it */

        if(info->current_box->probench_box.box_type==SEQUENCE)
           {
            blp=info->current_box->children;
            info->current_box=blp->box;
            info->last_box_num=blp->box->box_number;
            return(get_next_op(info));
           }
         }
       else
           /* if none then work back to find the next box and
              check it for ops */
           if(next_box_from(info))
            return(get_next_op(info));
           else
             return(0);
      }
     return(0);
}

/*==========================================================*
 * next_op uses the op_num of the run to find the next op *
 * and if there are none then return 0                    *
 *==========================================================*/
int next_op(RUN_INFO *info)
{
    int next=info->op_num+1;
    OPLIST *box_ops;
    int loop;

    box_ops=info->current_box->op;

    for(loop=1;loop<next;loop++)
       {
        box_ops=box_ops->next_op;
        if(box_ops==NULL)
          {
           info->op_num=0;
```

```
            return(0);
          }
        }

    info->op_num=loop;
    info->current_op=box_ops;
    return(loop);
}

/*=========================================================*
 * work out which box is the next one when moving back *
 *=========================================================*/
int next_box_from(RUN_INFO *info)
{
    BOX *parent;
    BOXLIST *boxlp;

    /* get the parent */
    parent=get_parent(info->current_box,info->current_tree);

    /* if the current is the root then we are finished */
    if(info->current_box==info->current_tree->root)
        return(NULL);

    /* if the parent is an iteration then set the current
       to be the parent */
    if(parent->probench_box.box_type==ITERATION)
       {
        info->current_box=parent;
        info->last_box_num=info->current_box->box_number;
        return(1);
       }

    /* if the parent is a selection then set the current to be
       the parent and find the next box back from then */
    if(parent->probench_box.box_type==SELECTION)
       {
        info->current_box=parent;
        return(next_box_from(info));
```

```
      }

    /* the parent is a sequence then compare the child box
       numbers with the current box number until we either
       run out of boxes or we find a box with a number
       greater than the current.
       if we find a box greater then set it to be current */

    if(parent->probench_box.box_type==SEQUENCE)
       {
        boxlp=parent->children;

        while(boxlp!=NULL)
          {
           if(boxlp->box->box_number > info->current_box->box_number)
            {
             info->current_box=boxlp->box;
             return(1);
            }
           boxlp=boxlp->link;
          }

        /* no good children so set current to be the parent */
        info->current_box=parent;
        return(next_box_from(info));
       }
}
```

G.2 Diagrammer Source Code

G.2.1 Diagrammer Structures

```
/*==================================================*
 *                STRUCTS.H                         *
 * all the structures associated with the diagrammer *
 *==================================================*/

#ifndef JSPDIAG
#define JSPDIAG

#include "profile.h"


 struct OPLISTALS{
                  char op_type;
                  int op_num;
                  struct OPLISTALS *next_op;
                 };

 typedef struct OPLISTALS OPLIST;


typedef struct {
                int box_number;
                char box_name[10];
                int box_type;
                int no_children;
                OPLIST *op;
                struct BOXYSTRUCT *children;
                BOX_REC probench_box;
               }BOX;

 struct BOXYSTRUCT{
                   BOX *box;
                   struct BOXYSTRUCT *link;
                  };
```

```
typedef struct BOXYSTRUCT BOXLIST;

typedef struct {
                int total_boxes;
                BOX *root;
               }TREE;

#endif
```

G.2.2 Boxlist

```
/*=====================================================*
 *                    BOXLIST.H                        *
 * The formal spec for linked list used in boxlist     *
 * and the external declarations of the functions used *
 *=====================================================*/

#ifndef BOXLISTDEFn
#define BOXLISTDEFn
#include <stdio.h>
#include "structs.h"
#include "box.h"

/*****************************************************************
*****************************************************************
sort            linked_list
based on        int,type
  (*      type * linked_list -> linked_list  *)


(c)    NEWLIST:                          ->      linked_list
(c)    ADDLIST:       type x linked_list  ->      linked_list
       DELETE_NODE:         linked_list ->
       DELETE_FROM_LIST:   type x linked_list   ->      linked_list
       ADD_AT:      int x type x linked_list    ->      linked_list
       APPEND:      linked_list x linked_list   ->      linked_list
       POSITION_IN: type x linked_list      ->      linked_list
       OVERWRITE_AT:int x type x linked_list  ->      linked_list
       JOIN_LISTS:  linked_list x linked_list x int -> linked_list
(*      currently not implemented *)
       GET_AT:      int x linked_list       ->      type
       UNITLIST:    type                    ->      linked_list
*****************************************************************
*****************************************************************/


extern BOXLIST *create_boxlist(void);
extern BOXLIST *delete_from_list(BOX *,BOXLIST *);
```

```
extern BOXLIST *join_box_lists(BOXLIST *,BOXLIST *,int);
extern int position_in(BOX *,BOXLIST *);
extern int pos_in(BOX *,BOXLIST *,int);
extern BOXLIST *append(BOXLIST *,BOXLIST *);
extern BOXLIST *add_at(int,BOX *,BOXLIST *);
extern BOXLIST *overwrite_at(int,BOX *,BOXLIST *);
extern void scrub_boxlist(BOXLIST *);

#endif
```

```
/*==============================================================*
 *          BOXLIST.C                                           *
 * This .C contains the most specified linked list used         *
 * it is one of the main workhorses of the diagrammer           *
 * and as such has been the most formally described.            *
 *                                                              *
 * No other linked list in the project undergoes so much        *
 * and so they have used the formal description presented here.*
 *==============================================================*/
#include <stdio.h>
#include "structs.h"
#include "box.h"
#include "boxlist.h"

/*****************************************************************
*****************************************************************
sort            linked_list
based on        int,type

    (* type * linked_list -> linked_list  *)


(c)   NEWLIST:              ->      linked_list
(c)   ADDLIST:  type x linked_list      ->      linked_list
      DELETE_NODE:            linked_list ->
      DELETE_FROM_LIST:       type x linked_list  ->      linked_list
      ADD_AT:         int x type x linked_list    ->      linked_list
      APPEND:         linked_list x linked_list   ->      linked_list
      POSITION_IN:            type x linked_list  ->      linked_list
      OVERWRITE_AT: int x type x linked_list  ->      linked_list
      JOIN_LISTS: linked_list x linked_list x int -> linked_list

(*      currently not implemented *)
      GET_AT:      int x linked_list     ->      type
      UNITLIST:    type        ->      linked_list
*****************************************************************
*****************************************************************/
```

```
/*****************************************************************************
JOIN_LISTS(ADDLIST(b1,l1),ADDLIST(b2,l2),pos)= if pos = 0
                        then ADDLIST(b1,APPEND((b2,l2),l1))
                        else ADDLIST(b1,JOIN_LISTS(l1,(b2,l2),pos));
*****************************************************************************/
BOXLIST *join_box_lists(BOXLIST *boxlist1,BOXLIST *boxlist2,int position)
  {
    if(position==0)
       {
         boxlist1->link=append(boxlist2,boxlist1->link);
         return boxlist1;
       }
    else
       {
         boxlist1->link=join_box_lists(boxlist1->link,
                                       boxlist2,position-1);
         return boxlist1;
       }
  }

/*****************************************************************************
POSITION_IN(b,ADDLIST(t,l))=    POS_IN(b,ADDLIST(t,l),0);
POS_IN(b,ADDLIST(t,l),pos)      =       if b=t
                                        then pos
                                        else POS_IN(b,l,pos+1);
*****************************************************************************/
int position_in(BOX *box,BOXLIST *boxlist)
       {
               return pos_in(box,boxlist,0);
       }

int pos_in(BOX *box,BOXLIST *boxlistpointer,int pos)
       {
               if(box==boxlistpointer->box)
                       return pos;
               else
                       pos_in(box,boxlistpointer->link,pos+1);
       }
```

```
/*****************************************************************
APPEND(ADDLIST(b1,l1),ADDLIST(b2,l2))   =       ADDLIST(b1,
                                                APPEND(l1,ADDLIST(b2,l2)));
APPEND(ADDLIST(b1,NEWLIST),ADDLIST(b2,l2))=     ADDLIST(b1,ADDLIST(b2,l2));
*****************************************************************/
BOXLIST *append(BOXLIST *boxlist1,BOXLIST *boxlist2)
  {
     if(boxlist1->link==NULL)
         {
           boxlist1->link=boxlist2;
           return boxlist1;
         }
     else
         {
           boxlist1->link=append(boxlist1->link,boxlist2);
           return boxlist1;
         }
  }

/*****************************************************************
DELETE_FROM_LIST(b,NEWLIST) = NEWLIST;
DELETE_FROM_LIST(b,ADDLIST(t,l))        = if b=t then l
                                     else ADDLIST(DELETE_FROM_LIST(b,l));
*****************************************************************/
BOXLIST *delete_from_list(BOX *box,BOXLIST *boxlistpointer)
   {
    if(boxlistpointer==NULL)
         return NULL;

    if(box==boxlistpointer->box)
        {
          scrub_box(box);
          scrub_boxlist(boxlistpointer);
          return boxlistpointer->link;
        }
    else
        {
          boxlistpointer->link=delete_from_list(box,
```

```
                                        boxlistpointer->link);
          return boxlistpointer;
        }
    }

/*****************************************************************
ADD_AT(pos,b,NEWLIST)   = ADDLIST(b,NEWLIST);
ADD_AT(pos,b,ADDLIST(t,l))      = if pos=0  then ADDLIST(b,ADDLIST(t,l))
                                            else ADD_AT(pos-1,b,l);
*****************************************************************/
BOXLIST *add_at(int position,BOX *box,BOXLIST *boxlistpointer)
   {
     BOXLIST *newboxlist;

     if(boxlistpointer==NULL)
       {
         newboxlist=create_boxlist();
         newboxlist->box=box;
         newboxlist->link=NULL;
         return newboxlist;
       }
     else
       {
        if(position==0)
           {
             newboxlist=create_boxlist();
             newboxlist->box=box;
             newboxlist->link=boxlistpointer;
             return newboxlist;
           }
        else
           {
             boxlistpointer->link=add_at(position-1,
                                         box,
                                         boxlistpointer->link);
             return boxlistpointer;
           }
       }
  }
```

```
/*********************************************************************
OVERWRITE_AT(pos,b1,ADDLIST(b2,l2)) = if pos=0
                                      then ADDLIST(b1,l2);
                                 else ADDLIST(b2,overwrite_at(pos-1,b1,l2));
*********************************************************************/
BOXLIST *overwrite_at(int position,BOX *box,BOXLIST *boxlistpointer)
   {
     if(position==0)
       {
         boxlistpointer->box=box;
         return boxlistpointer;
       }
     else
       {
         boxlistpointer->link=overwrite_at(position-1,
                                           box,
                                           boxlistpointer->link);
         return boxlistpointer;
       }
  }

/*************************************************************
(c)              NEWLIST:    ->      linked_list
*************************************************************/
BOXLIST *create_boxlist()
{
    BOXLIST *list;

    if((list=(BOXLIST *)malloc(sizeof(BOXLIST)))==NULL)
           fatal_error("no more room for boxlist");

    list->box=NULL;
    list->link=NULL;
    return list;
}

/*********************************************
                DELETE_NODE:    linked_list ->
```

```
*****************************************/
void scrub_boxlist(boxlist)
        BOXLIST *boxlist;
{
        free(boxlist);
}

/*****************************************************
       constructor for a box list with a supplied box
*****************************************************/
BOXLIST *unit_box_list(BOX *box)
{
  BOXLIST *list;

  if((list=(BOXLIST *)malloc(sizeof(BOXLIST)))==NULL)
        fatal_error("no more room for boxlist");

  list->box=box;
  list->link=NULL;
  return list;
}
```

G.2.3 Boxes

```
/*=================================================*
 *              BOX.H                              *
 * external definitions of the functions that act on *
 * the box structure of the diagrammer              *
 *=================================================*/

#ifndef BOXF
#define BOXF
#include <stdio.h>
#include "structs.h"

/********
*********       FUNCTION DEFINITIONS
*********/
extern void box_print(BOX *);
extern void name_box(BOX *,char *);
extern void make_root(BOX *);
extern BOX *create_box(void);
extern void scrub_box(BOX *);
extern void edit_box(BOX *,TREE *);


#endif
```

```
/*=================================*
 *                BOX.C            *
 * The functions that act on the box *
 * structure of the diagrammer      *
 *=================================*/
#include <stdio.h>
#include "structs.h"
#include "treebox.h"
#include "box.h"


/*==================================================*
*  Print a box, mainly prints the non Probench     *
*  Features, but prints out the probench box name  *
*==================================================*/
void box_print(box)
        BOX *box;
{
    int loop;
    BOXLIST *listpointer;
    BOX *boxpointer;
    OPLIST *oppointer;

    printf("BOX number %d ",box->box_number);

    printf("type:- %d \n n.o children:- %d\n",box->box_type,
                                            box->no_children);

    printf("name:- \n");
    for(loop=0;loop<10;loop++)
         printf("%c",box->probench_box.box_name[0][loop]);
    for(loop=0;loop<10;loop++)
         printf("%c",box->probench_box.box_name[1][loop]);


    if((box->no_children==0)&&((box->op)==NULL))
        printf(" box has not been allocated anything\n");
    else
        if(box->no_children==0)
```

```
            {
             printf("ops:- ");
             oppointer=box->op;
             while(oppointer->next_op!=NULL)
                {
                 printf("%d ",oppointer->op_num);
                 oppointer=oppointer->next_op;
                }
             printf("%d.",oppointer->op_num);
             printf("\n");
            }

    if((box->no_children)!=0)
      {
        listpointer=box->children;
        boxpointer=listpointer->box;
        printf("children list:- %d",boxpointer->box_number);
        while(listpointer->link!=NULL)
           {
            listpointer=listpointer->link;
            boxpointer=listpointer->box;
            printf(",%d",boxpointer->box_number);
           }
        printf("\n");
      }
}


/*=====================================*
 * Make the supplied box the root box *
 *=====================================*/
void make_root(BOX *box)
{
   box->box_number=1;
   name_box(box,"root box  ");
   box->box_type=1;
   box->no_children=0;
   box->op=NULL;
   box->children=NULL;
```

```
}

/*=====================================*
 * make the supplied name the box name *
 * the name is placed in the probench  *
 * information side                    *
 *=====================================*/
void name_box(box,name)
        BOX *box;char *name;
{
        strcpy(box->probench_box.box_name,name);
}

/*=====================================*
 * Delete the supplied box and the     *
 * corresponding operations            *
 *=====================================*/
void scrub_box(BOX *box)
{
        OPLIST *oplisty,*oplisty2;

        free(box);

        oplisty=box->op;        /* point to first op */
        while(oplisty!=NULL)    /* do until no more ops */
                {
                 oplisty2=oplisty->next_op;  /* get next op */
                 free(oplisty);              /* delete current op */
                 oplisty=oplisty2;           /* point to next op */
                }
}

/*=============================================*
 * Create a box and return the box address     *
 * the default value allocations are mainly    *
 * for the diagrammer details because the      *
 * probench values are intended to be loaded   *
 * from file, the default box_type was added   *
```

```
 * because no probench .pr0 file is read in   *
 *==========================================*/
BOX *create_box(void)
{
        BOX *allocated;

        if((allocated=(BOX *)malloc(sizeof(BOX)))==NULL)
                fatal_error("no more room for box");

        allocated->box_number=0;
        allocated->box_type=1;
        allocated->no_children=0;
        allocated->op=NULL;
        allocated->children=NULL;

        allocated->probench_box.box_type=SEQUENCE;

        return allocated;
}


/*==============================================*
 * A simple box editing function for the test   *
 * diagrammer, allows the setting of a box type *
 * the addition of operations, and conditions   *
 * , operations are appended to the operations  *
 * list                                         *
 *                                              *
 * NO CHECKS made on the existance of operations*
 * or conditions.                               *
 *==============================================*/
void edit_box(BOX *box,TREE *tree)
   {
      int num=0;
      OPLIST *opl;
      BOX *parent;

      parent=get_parent(box,tree); /* get the parent */
```

```
/* show the type menu and get response */

   printf("box is type %d\n",box->probench_box.box_type);
   printf("new type is :\n");
   printf("%d. sequence\n%d. selection\n%d. iteration\n => ",
          SEQUENCE,SELECTION,ITERATION);
   scanf("%d",&num);
   box->probench_box.box_type=num;

 /* if parent was selection or iteration then the box
    needs a condition, assuming it is not the root of
    course */

 if((parent->probench_box.box_type==SELECTION ||
     parent->probench_box.box_type==ITERATION) &&
     parent != box)
    {
     printf("\nbox has cond number %d\n",box->probench_box.cond_num);
     printf("new cond is => ");
     scanf("%d",&(box->probench_box.cond_num));
    }

 /* if box does not have children then operations may be added */
 if(box->no_children==0)
    {
     printf("\nadd ops ... 0 to stop => ");

     scanf("%d",&num);
     op1=box->op;

     while(num!=0)
       {
        if(op1==NULL)    /* if it is the first op */
          {              /* then create and add   */
           box->op=create_op();
           op1=box->op;
           op1->op_num=num;
          }
        else             /* otherwise loop through ops */
```

```
            {               /* and place it on the end    */
             while(opl->next_op!=NULL)
                opl=opl->next_op;

             opl->next_op=create_op();
             opl=opl->next_op;
             opl->op_num=num;
            }
          printf("=> ");
          scanf("%d",&num);  /* get next */
         }
      }
}
```

G.2.4 Operations

```
/*=============================================*
 *              OP.H                           *
 *                                             *
 * This file contains the declarations for     *
 * for the jsp operations                      *
 *=============================================*/

#ifndef OP
#define OP
#include <stdio.h>
#include "structs.h"

/*******
********        FUNCTION DECLARATIONS
********/

extern OPLIST *create_op(void);

#endif
```

```
/*==========================================
 *            OP.C                          *
 *                                          *
 * This file contains the functions         *
 * for the jsp operations on the diagrammer *
 *==========================================*/
#include "op.h"



/*******
********        FUNCTIONS
********/

/*=====================================*
 * create_op allocates space for an op *
 *=====================================*/
OPLIST *create_op()
{
        OPLIST *opy;

        if((opy=(OPLIST *)malloc(sizeof(OPLIST)))==NULL)
                        fatal_error("no more room for oplist");

        opy->op_num=0;
        opy->next_op=NULL;

        return opy;
}
```

G.2.5 Tree Structure

```
/*=========================================*
 *                TREE.H                   *
 * definitions of tree specific functions *
 *=========================================*/

#ifndef TREEDEF
#define TREEDEF

#include "structs.h"
#include "boxlist.h"
#include <stdio.h>
/*******
                FUNCTION DEFINITIONS
********/

extern TREE *create_tree(void);

#endif
```

```
/*==============================*
 *          TREE.C              *
 * the tree specific functions *
 *==============================*/

#include "tree.h"


/********
*********       FUNCTIONS
*********/

TREE *create_tree()
{
        TREE *tree;

        if((tree=(TREE *)malloc(sizeof(TREE)))==NULL)
                fatal_error("no more room for tree");

        tree->total_boxes=1;
        tree->root=create_box();
        make_root(tree->root);
        return tree;
}
```

G.2.6 Tree Manipulation

```
/*=========================================================*
 *                TREEBOX.H                                *
 * contains all the functions relating to trees and boxes *
 * that were too general to fit into tree.h or box.h      *
 *=========================================================*/

#ifndef BOXFns
#define BOXFns

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

/********
*********        FUNCTION DEFINITIONS
*********/

extern void renumber_from(TREE *);
extern int renumber(BOX *,int);

extern BOX *find_box(int,TREE *);
extern BOX *find(int,BOX *);

extern BOX *create_box(void);

extern void create_under(int,TREE *);
extern void add_under(BOX *,BOX *,TREE *);

extern void create_right(int,TREE *);
extern void add_right(BOX *,BOX *,TREE *);

extern void create_left(int,TREE *);
extern void add_left(BOX *,BOX *,TREE *);

extern void create_above(int,TREE *);
extern void add_above(BOX *,BOXLIST *,TREE *);

extern BOX *get_parent(BOX *,TREE *);
```

```
extern BOX *getparent(BOX *,BOX *,TREE *);

extern BOXLIST *get_boxlist(BOX *,TREE *);

extern void delete(int,TREE *);
extern void delbox(BOX *,TREE *);

#endif
```

```
/*=====================================*
 *        TREEBOX.C                    *
 * contains all tree and box functions *
 *=====================================*/

#include "treebox.h"

/********
*********       FUNCTIONS
*********/

/*=========================================================*
 * find_box initiates a call to the recursive find with *
 * the root box as the first box to check               *
 *=========================================================*/
BOX *find_box(box_num,tree)
        int box_num;TREE *tree;
{
        if (tree->total_boxes<box_num)
                error("box does not exist");

        return (BOX *)find(box_num,tree->root);
}

/*==============================================================*
 * find checks the box number being sought with the box    *
 * number of the box pointed to, if they are the same then *
 * return the address of the box, otherwise check for      *
 * children and call find with each of the children until  *
 * the box is found                                        *
 *==============================================================*/
BOX *find(box_num,root)
        int box_num;BOX *root;
{
        BOXLIST *childpointer=root->children;
        BOX             *returned=NULL;

        if(root->box_number==box_num)
                        return root;
```

```
        else
           if(root->no_children==0)
                   return NULL;
           else
             {
                /* check the first child */

                returned=find(box_num,childpointer->box);

                /* if the box has not been found then loop around
                   until we run out of children */

                while((returned==NULL) && (childpointer->link!=NULL))
                    {
                      childpointer=childpointer->link;
                      returned=find(box_num,childpointer->box);
                    }

                /* either the box is found or not, but returned
                   has been set either way so return it */

                return returned;
              }
}

/*==========================================================*
 * renumber_from initiates the recursive renumber function *
 *==========================================================*/
void renumber_from(TREE *tree)
{
        int dummy;
        dummy=renumber(tree->root,0);
}

/*==================================================================*
 * renumber, causes the tree to be renumbered, it is used when     *
 * boxes are added or deleted, it is passed the current box, and *
 * the number of the last box, the current box is set to be the    *
 * last box number + 1, the function then cycles through the       *
```

```
 * children of the current box causing them and their children   *
 * to be renumbered                                              *
 *===============================================================*/
int renumber(current,last)
               BOX *current;int last;
{
        BOXLIST *children_of;
        BOX             *boxpointer;

        last++;
        current->box_number=last;

        if(current->no_children!=0)
                {
                        children_of=current->children;
                        boxpointer= children_of->box;
                        last=renumber(boxpointer,last);

                        while(children_of->link!=NULL)
                                {
                                        children_of=children_of->link;
                                        boxpointer=children_of->box;
                                        last=renumber(boxpointer,last);
                                }
                }
        return last;
}


/*===============================================================*
 * this function creates a new box under another, increases the *
 * total number of boxes, and renumbers the tree                 *
 *===============================================================*/
void create_under(box_num,tree)
               int box_num;TREE *tree;
{
        add_under(find_box(box_num,tree),create_box(),tree);
        tree->total_boxes++;
```

```
        renumber_from(tree);

}

/*==================================================================*
 * add under gives the parent ops to the child, assuming the       *
 * child has none, the parent's children pointer is given to this *
 * child box and the child is added to the parent's children       *
 *==================================================================*/
void add_under(BOX *parent,BOX *child,TREE *tree)
        {
                child->box_type=1;
                child->no_children=parent->no_children;
                if(child->op==NULL)
                     child->op=parent->op;
                child->children=parent->children;
                parent->op=NULL;
                parent->children=NULL;
                parent->children=add_at(0,child,parent->children);
                parent->no_children=1;
        }


/*=========================================================*
 * get_boxlist, gets the list link that the box resides in *
 * to do this all the child links of the parent are scanned*
 * until the link containing the box is found              *
 *=========================================================*/
BOXLIST *get_boxlist(box,tree)
                BOX *box;TREE *tree;
{
                BOXLIST *boxlistpointer;
                BOX     *parent;

                parent=get_parent(box,tree);

                boxlistpointer=parent->children;

                while(boxlistpointer->box!=box)
```

```
                        boxlistpointer=boxlistpointer->link;

                return boxlistpointer;
}

/*==============================================================*
 * get_parent initates a call to the recursive function getparent *
 *==============================================================*/
BOX *get_parent(target,tree)
                BOX *target;TREE *tree;
{
                return getparent(tree->root,target,tree);
}

/*==============================================================*
 * getparent checks the target box to see if it is the root box *
 * if so then for the purposes of the program it is its own     *
 * parent, then the box is checked for children. if there are no*
 * children then return null, otherwise work through them       *
 * checking the child boxes to see if they have the target as a *
 * child                                                        *
 *==============================================================*/
BOX *getparent(current,target,tree)
                BOX *current,*target;TREE *tree;
{
                BOXLIST *boxlistpointer=current->children;
                BOX     *returned=NULL;

        if(target->box_number==1)
              return(target);

        if((current->no_children!=0)&&(boxlistpointer->box == target))
             return current;
        if(current->no_children==0)
                return NULL;
        else
            {
             returned=getparent(boxlistpointer->box,target,tree);
```

```
            while((returned==NULL)&&(boxlistpointer->link!=NULL))
               {
                   boxlistpointer=boxlistpointer->link;
                   if((current->no_children!=0)&&
                      (boxlistpointer->box == target))
                         return current;

                   returned=getparent(boxlistpointer->box,target,tree);
               }
            return returned;
           }
}

/*================================================================*
 * initates a call to add right provided we are not adding to the *
 * right of the root box                                          *
 *================================================================*/
void create_right(box_num,tree)
               int box_num;TREE *tree;
{
               if(box_num==1)
                      error("cannot add right of root box");
               else
                 {
                   add_right(find_box(box_num,tree),create_box(),tree);
                   tree->total_boxes++;
                   renumber_from(tree);

                 }
}

/*=========================================================*
 * add right first finds the position in the list of the *
 * box we are trying to add to the right of, we then add *
 * the box to the list at this position + 1              *
 *=========================================================*/
void add_right(lefty,righty,tree)
               BOX *lefty,*righty;TREE *tree;
{
```

```
        BOX *parent=get_parent(lefty,tree);

        parent->children=add_at(position_in(lefty,parent->children)+1,
                                                        righty,
                                                        parent->children);
        parent->no_children++;
}

/*================================================================*
 * initates a call to add left provided we are not adding to the  *
 * left of the root box                                            *
 *================================================================*/
void create_left(box_num,tree)
                int box_num;TREE *tree;
{

                if(box_num==1)
                        error("cannot add left of root box");
                else
                  {
                        add_left(create_box(),find_box(box_num,tree),tree);
                        tree->total_boxes++;
                        renumber_from(tree);

                  }
}

/*=====================================================*
 * add left first finds the position in the list of the *
 * box we are trying to add to the left of, we then add *
 * the box to the list at this position.                *
 * add_at is not destructive and inserts the box        *
 *=====================================================*/
void add_left(lefty,righty,tree)
                BOX *lefty,*righty;TREE *tree;
{
        BOX *parent=get_parent(righty,tree);
```

```
        parent->children=add_at(position_in(righty,parent->children),
                                                        lefty,
                                                        parent->children);

        parent->no_children++;

}

/*==============================================================*
 * initates a call to add above provided we are not adding    *
 * above the root box                                          *
 *==============================================================*/
void create_above(box_num,tree)
                int box_num;TREE *tree;
{
        if(box_num==1)
             error("cannot add above root box");
        else
          {
           add_above(create_box(),get_boxlist(find_box(box_num,tree),tree),tre
           tree->total_boxes++;
           renumber_from(tree);
          }
}

/*==================================================================*
 * add_above adds the lower box to the upper box's children list *
 * then the parent box of the lower box has the new box as a     *
 * child.                                                         *
 * effectively a box is added vertically between two boxes        *
 *==================================================================*/
void add_above(BOX *upy,BOXLIST *downy,TREE *tree)
       {
               BOX *parent=get_parent(downy->box,tree);

               upy->children=add_at(0,downy->box,upy->children);
               upy->no_children=1;

               parent->children=overwrite_at(position_in(downy->box,
```

```
                                                parent->children),
                                                upy,
                                               parent->children);
        }


/*==========================================================*
 * delete calls the delete box function provided we are not *
 * trying to delete the root box                            *
 *==========================================================*/
void delete(box_num,tree)
                int box_num;TREE *tree;
{
        if(box_num==1)
                error("cannot delete root box");
        else
                delbox(find_box(box_num,tree),tree);
}

/*==========================================================*
 * delbox uses the boxlist's delete function to remove a box *
 * from the list of children, if the box to be deleted has   *
 * children then these are inserted to the parents children  *
 * list at the same position of the deleted box              *
 *==========================================================*/
void delbox(BOX *box,TREE *tree)
        {
                BOX *parent=get_parent(box,tree);

                if(box->children==NULL)
                  parent->children=delete_from_list(box,parent->children);
                else
                   {
                     join_box_lists(parent->children,box->children,
                                    position_in(box,parent->children));
                     parent->no_children+=box->no_children;
                     parent->children=delete_from_list(box,parent->children);
                   }
                parent->no_children--;
```

```
        renumber_from(tree);
    }
```

G.2.7 Diagrammer Functions

```
/*=================================*
 *        SYSTEM.H                 *
 * the system functions            *
 *=================================*/

#ifndef SYSTEM
#define SYSTEM

#include "structs.h"
#include "drawprim.h"
#include "treebox.h"
#include "alpro.h"
#include <stdio.h>

/*******
********     FUNCTION DEFINITIONS
********/
extern void prntree(BOX *,TREE *);
extern void parse(TREE *);
extern void print_sizes(void);
extern void draw_tree(BOX *,int,int);
extern void draw_from(BOX *,int,int,int *,int *,int,int,int,int);


#endif
```

```
/*==========================================================*
 *                  SYSTEM.C                                *
 * these functions are all system related to the diagrammer *
 * the largest being parse() which was developed in a slowly*
 * built up manner and now needs to be redone, draw_tree    *
 * also uses a primitive algorithm                          *
 *==========================================================*/
#include "system.h"
#include <osbind.h> /* ST operating system access calls */


/*******
********        FUNCTIONS
********/

/*==========================================================*
 * a test function to print the sizes of structures in use *
 *==========================================================*/
void print_sizes(void)
{
        printf("sizes of structures\n");
        printf("OPLIST %d bytes\n",sizeof(OPLIST));
        printf("BOX    %d bytes\n",sizeof(BOX));
        printf("BOXLIST %d bytes\n",sizeof(BOXLIST));
        printf("TREE    %d bytes\n",sizeof(TREE));
        printf("press return to continue \n");
        getchar();
}

/*==========================================================*
 * a recursively defined function to print the tree in text *
 * adapted from the function given in programs and data     *
 * structures by Leendert Ammeraal to print b-trees         *
 * although the one here is more sophisticated              *
 *==========================================================*/
void prntree(box,tree)
        BOX *box;TREE *tree;
{
        BOXLIST *children_of;
```

```
        OPLIST  *ops;
        BOX             *boxpointer,*parent;
        static indent=0;            /* this is static and used
                                   by all calls to prntree */
        int loop;
        char type=' ';

        parent=get_parent(box,tree); /* get the parent */

 if(box->box_number!=1)  /* so long as its not the root */
   {
        if(parent->probench_box.box_type==SELECTION)
             type='o';
        else{ if(parent->probench_box.box_type==ITERATION)
                  type='*';
             else type=' ';
             }
   }

        indent+=4; /* indent the box */

        if(box->no_children==0) /* if box has no children */
            {
             for(loop=0;loop<indent;loop++) /* indent */
                   printf(" ");
             printf("%c%d",type,box->box_number); /* print the type
                                                   and box num */
             ops=box->op;

             /* now print the ops*/
             if(ops!=NULL)
                {
                  printf(" O-%d,",ops->op_num);
                  while(ops->next_op!=NULL)
                     {
                      ops=ops->next_op;
                      printf("%d,",ops->op_num);
                     }
                }
```

```
            printf("\n");
        }
    else        /* box has children so print box then children */
        {
         for(loop=0;loop<indent;loop++)
                printf(" ");
         printf("%c%d\n",type,box->box_number);
         children_of=box->children;
         boxpointer=children_of->box;
         prntree(boxpointer,tree);

         while(children_of->link!=NULL) /* loop around children */
              {
                children_of=children_of->link;
                boxpointer=children_of->box;
                prntree(boxpointer,tree);
              }
        }
    indent-=4; /* de-indent */
}

/*==========================================================*
 * a large function which control the diagrammer, in fact *
 * it is basically the diagrammer, it loads in the        *
 * probench operations, conditions, and symbol table      *
 * for a hard coded diagram, it is basically a test bed   *
 *==========================================================*/
void parse(TREE *current_tree)
{
        char command[2];
        int x,y,size;
        int box_num;
        int multiple;
        int loop,op_stat;
        char name[20];
        RUN_INFO *run;
        SYMTAB_ENTRY **symtab=create_symbol_table_array(100);
        OP_LIST_LIST **op_array=create_op_array();
```

```
COND **cond_array=create_cond_array();

/* load the details */
load_symtab(symtab,"b:symtable.out");
cross_reference(symtab,100);
load_cond_array("b:cndout2.dat",cond_array,symtab);
load_ops_array("b:opsarray.out",op_array,symtab);

run=create_run(current_tree,symtab,cond_array,op_array);

printf("\n");
prntree(current_tree->root,current_tree);
printf("\n Command , ?? -help:");
scanf("%s",command);
run->finished=0;
multiple=0;

/* loop around getting 2 character commands and
   any necessary details */

while(strcmp(command,"xx"))
   {
     if(!strcmp(command,"??"))
          {
              printf("\n  COMMANDS FOR ICD:\n");
              printf("au - add under\n");
              printf("al - add left\n");
              printf("ar - add right\n");
              printf("aa - add above\n");
              printf("mu - add multiple under , needs total n.o boxes\
              printf("ml - add multiple left , needs total n.o boxes\n
              printf("mr - add multiple right , needs total n.o boxes\
              printf("ma - add multiple above , needs total n.o boxes\
              printf("pr - print box\n");
              printf("nm - name box , needs name\n");
              printf("sh - show tree\n");
              printf("dd - delete box number\n");
              printf("db - draw box x , y , size , testing only\n");
              printf("dt - draw tree\n");
```

```
        printf("eb - edit box number\n");
        printf("do - display ops,  dc - display conds\n");
        printf("ds - display symbols, rr - reset run\n");
        printf("ss - single step , rc -run to completion\n");

        printf("xx - exit\n");
        printf("?? - command list\n");
       }
    else

  /* single step calls the next op and prints the op
     unless the run is finished */

   if(!strcmp(command,"ss"))
     {
      if(run->finished)
         printf("\n ******** PROGRAM END *********\n");
      else{
           if(!get_next_op(run))
              {
                run->finished=1;
                printf("\n ******** PROGRAM END *********\n");
              }
           else{
              printf("BOX  %d \n",run->current_box->box_number);
              printf("OP   %d \n",run->current_op->op_num);
              display_operation(run->current_ops
                                [run->current_op->op_num]);
              evaluate_operation(run->current_ops
                                 [run->current_op->op_num]);
                }

           }
     }
  else

  /* run to completion evaluates ops in turn
     with only displays by the program
     showing up*/
```

```
if(!strcmp(command,"rc"))
   {
     if(run->finished)
        printf("\n ******** PROGRAM END *********\n");
     else{
          op_stat=get_next_op(run);
          while(op_stat && (!Bconstat(2)))
             {
             evaluate_operation(run->current_ops
                                [run->current_op->op_num]);
             op_stat=get_next_op(run);
             }
          if(op_stat==0)
            {
             run->finished=1;
             printf("\n ******** PROGRAM END *********\n");
            }
         }
    }
else
  /* reset the run */
if(!strcmp(command,"rr"))
      {
         reset_run(run);
       }
else
  /* display the symbol table */
if(!strcmp(command,"ds"))
     {
        quick_display_symtab(symtab,100);
     }
else
  /* display the conditions */
if(!strcmp(command,"dc"))
       {
        display_conditions(cond_array);
       }
else
```

```
  /* display the operations */
if(!strcmp(command,"do"))
     {
          display_operations(op_array);
     }
else
  /* add a box under  the specified box */
if(!strcmp(command,"au"))
     {
      scanf("%d",&box_num);
      create_under(box_num,current_tree);
     }
else
/* add under box a number of boxes */
if(!strcmp(command,"mu"))
     {
      scanf("%d%d",&box_num,&multiple);
      create_under(box_num,current_tree);
      for(loop=1;loop<multiple;loop++)
            create_right(box_num+1,current_tree);
     }
else
/* add a box left of the desired box */
if(!strcmp(command,"al"))
     {
      scanf("%d",&box_num);
      create_left(box_num,current_tree);
     }
else
/* add a number of boxes left*/
if(!strcmp(command,"ml"))
     {
      scanf("%d%d",&box_num,&multiple);
      create_left(box_num,current_tree);
      for(loop=1;loop<multiple;loop++)
            create_left(box_num,current_tree);
     }
else
/* add a box right */
```

```
if(!strcmp(command,"ar"))
     {
      scanf("%d",&box_num);
      create_right(box_num,current_tree);
     }
else
/* add a number of boxes left */
if(!strcmp(command,"mr"))
      {
       scanf("%d%d",&box_num,&multiple);
       create_right(box_num,current_tree);
       for(loop=1;loop<multiple;loop++)
               create_right(box_num,current_tree);
      }
else
/* add a box above */
if(!strcmp(command,"aa"))
      {
       scanf("%d",&box_num);
       create_above(box_num,current_tree);
      }
else
/* add a number of boxes above */
if(!strcmp(command,"ma"))
     {
       scanf("%d%d",&box_num,&multiple);
       create_above(box_num,current_tree);
       for(loop=1;loop<multiple;loop++)
            create_above(box_num,current_tree);
     }
else
/* print box details for a pecific box */
if(!strcmp(command,"pr"))
     {
      scanf("%d",&box_num);
      box_print(find_box(box_num,current_tree));
     }
 else
/* name a box */
```

```
 if(!strcmp(command,"nm"))
      {
       scanf("%d",&box_num);
       scanf("%s",&name);
       name_box(find_box(box_num,current_tree),name);
      }
  else
/* display the tree textually */
  if(!strcmp(command,"sh"))
      {
        printf("\n");
        prntree(current_tree->root,current_tree);
      }
  else
/* test the draw box function */
  if(!strcmp(command,"db"))
      {
        scanf("%d",&x);
        scanf("%d",&y);
        scanf("%d",&size);
        draw_box(x,y,size);
      }
  else
/* delete a box */
  if(!strcmp(command,"dd"))
      {
        scanf("%d",&box_num);
        delete(box_num,current_tree);
      }
  else
/* draw the tree graphically */
  if(!strcmp(command,"dt"))
      {
       draw_tree(current_tree->root,10,10);
      }
  else
/* edit the box details */
  if(!strcmp(command,"eb"))
      {
```

```
                    scanf("%d",&box_num);
                    edit_box(find_box(box_num,current_tree),current_tree);
                  }
               else
                  printf("command not recognised\n");


               printf("\n Command ?? -help:");
               scanf("%s",command);


     }
}

/*=========================================================*
 * draw tree starts off the recursive draw_from function *
 *=========================================================*/
void draw_tree(BOX *box,int xlimit,int ylimit)
        {
        int currx=200,curry=200;

        draw_from(box,xlimit,ylimit,&currx,&curry,0,0,0,0);
        }

/*=========================================================*
 * draw_from draws a box at the current x and y and calls *
 * draw_from with the child boxes                          *
 *=========================================================*/
void draw_from(BOX *box,int xlimit,int ylimit,int *currx,
               int *curry,int xcount,int ycount,int parentx,int parenty)
        {
                BOXLIST *children_of;
                BOX *boxpointer;
                int size=10;                    /* small test values only */
                int log_box=20;
                int spx,spy;

                ycount++;
```

```
        *curry+=log_box;  /* advance the y co-ord */

        if(box->no_children!=0) /* if box has children */
                {

/* first check to see if the box is the root,
   if not then draw a line from the child to the parent */
   if(parentx!=0 && parenty!=0)
     draw_line((parentx+(size/2)),parenty+size,
               (*currx+(size/2)),*curry);

        /* draw the box */
        draw_box(*currx,*curry,size);

        /* save x and y for return */
        spx=*currx;spy=*curry;

        /* do first child */
        children_of=box->children;
        boxpointer=children_of->box;
        draw_from(boxpointer,xlimit,ylimit,currx,curry,
                  xcount,ycount,spx,spy);

        /* do rest of children */
        while(children_of->link!=NULL)
           {
             children_of=children_of->link;
             boxpointer=children_of->box;
             draw_from(boxpointer,xlimit,ylimit,currx,
                       curry,xcount,ycount,spx,spy);
           }
      }
else
      {
      /* if no children then increase the x co-ord for
        future use */
       if(parentx!=0 && parenty!=0)
             draw_line((parentx+(size/2)),parenty+size,
                       (*currx+(size/2)),*curry);
```

```
            draw_box(*currx,*curry,size);
            *currx+=log_box;
            xcount++;
          }
        *curry-=log_box;  /* decrease the y co-ord */
        ycount--;
}
```

G.2.8 Drawing Functions

```
/*==============================================================*
 *                    DRAWPRIM.H                                *
 * declares the global variables used by the drawing primitives *
 * and externally declares the primitives themselves            *
 *==============================================================*/

#ifndef DRAW
#define DRAW

#include <vdi.h>  /* the gem/ST drawing routines */

extern void start_gem(void);
extern void stop_gem(void);
extern void draw_line(int,int,int,int);
extern void draw_box(int,int,int);


/* atari st gem stuff */

int contrl[12];
int intin[128];
int ptsin[128];
int intout[128];
int ptsout[128];

short handle;
short workin[11]={1,1,1,1,1,1,1,1,1,1,2},workout[57];
short co_ord[10];

#endif
```

```
/*==================================================================*
 *                    DRAWPRIM.C                                    *
 * the system drawing primitives, they are declared                 *
 * here so that the drawing can be ported to any system             *
 * by changing the actual contents of the functions and the header *
 * this remains unfinished since there is no clear screen function *
 * and the jsp box is drawn as a square rather than a rectangle,    *
 * but development stopped early on in the project timescale        *
 *==================================================================*/

#include "drawprim.h"


/*=========================================================================*
 * The graphic system is initialised, in this case GEM on the ATARI ST *
 *=========================================================================*/
void start_gem()
{
        short junk;
        appl_init();
        handle=graf_handle(&junk,&junk,&junk,&junk);
        v_opnvwk(workin,&handle,workout);

}

/*=====================================*
 * the graphic system is closed down *
 *=====================================*/
void stop_gem()
{
        v_clrwk(handle);
        appl_exit();

}

/*=================================================*
 * a non relative line draw from (x1,y1) to (x2,y2) *
 *=================================================*/
void draw_line(int x1,int y1,int x2,int y2)
```

```
{
                co_ord[0]=x1;
                co_ord[1]=y1;
                co_ord[2]=x2;
                co_ord[3]=y2;
                v_pline(handle,2,co_ord);
}

/*=====================================*
 * draw a box at (x1,y1) of size size *
 *=====================================*/
void draw_box(x1,y1,size)
        int x1,y1,size;
{

        co_ord[0]=x1;
        co_ord[1]=y1;
        co_ord[2]=x1+size;
        co_ord[3]=y1;
        co_ord[4]=x1+size;
        co_ord[5]=y1+size;
        co_ord[6]=x1;
        co_ord[7]=y1+size;
        co_ord[8]=x1;
        co_ord[9]=y1;
        v_pline(handle,5,co_ord);
}
```

G.2.9 The Diagrammer Program

```
/*==========================================================*
 *                    TSTLOAD.C                             *
 * this is the testbed program for the diagrammer, it       *
 * creates a tree, starts the graphics environment manager *
 * and calls the parse routine                              *
 *==========================================================*/

#include "alpro.h"

void main()
{
  TREE *tree=create_tree();
  start_gem();
  parse(tree);
}
```

G.3 Probench '.pr0' File Handling

The following code specifies the format of the .pr0 file, and the functions which can act upon it. Because of system differences betweem the ST and a PC, these .pr0 loading functions only work on a PC. But it does build up a tree using the diagrammer code, so with a little work, implementations could be made available for both machines.

```
/* PROFILE.H - Probench file structure information
** (c) SCM ltd 1992
*/
#ifndef DFERT
#define DFERT
/* box_type values */
#define SEQUENCE    0
#define SELECTION   1
#define ITERATION   2
#define BACK_TRACK  3
#define SUB_TYPE    4
#define LEAF        5

/**/

#define HEADER       1
#define BOX_RECORD   2
#define TRAILER      3
#define OPS_OVERFLOW 4

#define OP_GRP_SIZE   16

#define OPS_LIST    0
#define COND_LIST   1
#define QUIT_LIST   2

/* box comment handling information */
```

```
#define COMMENT_LEN(comment)  (*((int *)(comment)))
#define COMM_INFO   sizeof(int)

struct header_rec {
 char rec_type;
 char diag_type;
 char data_flow[26];
 int  invert_count;
 int  box_tot;
 int  file_check_no;
 char const_checked;  /* flag indicating consistency checked */
 char spare[3];
};

struct box_rec{
 char rec_type;
 int  goback_num;
 char box_type;
 char box_name[2][10];
 char cond_type;
 int  cond_num;
 char data_item;
 char struct_indic;
 char page_indic;
 int  ops_depth;
 char op_indic;
 char subtree_no;
 int  comment_size;  /* size in bytes of comment text for box */
 char spare[3];
};

typedef struct box_rec BOX_REC;

struct trailer_rec{
 char rec_type;
 int goback_total;
};

typedef struct trailer_rec TRAILER_REC;
```

```
#define PRO_CHECK_NO  20860

struct box_ops{
 char op_type[OP_GRP_SIZE];
 char op_num[OP_GRP_SIZE][3];
 char q_level[OP_GRP_SIZE]; /* quit level 'Q','R'...'Z' */
 int op_count;
 struct box_ops *next_ops;
};

typedef struct box_ops BOX_OPS;

/* end PROFILE.H */

#endif
```

```
/*=========================================================*
 *                    ALPRO.H                              *
 * This header file #includes ALL the files necessary      *
 * for the test diagrammer, and externally declares the *
 * functions acting on the .pr0 file from probench         *
 * these functions unfortunately only work on a PC         *
 *=========================================================*/

#ifndef ALPROFILEH
#define ALPROFILEH

#include <stdio.h>
#include "errors.h"
#include "structs.h"
#include "op.h"
#include "box.h"
#include "boxlist.h"
#include "tree.h"
#include "treebox.h"
#include "system.h"
#include "profile.h"
#include "alutil.h"
#include "alfile.h"
#include "ops.h"
#include "conds.h"
#include "trace.h"
#include "symtab.h"

extern void load_profile(char *,TREE *);
extern int read_box(FILE *,BOX *);
extern int process_header(char *,struct header_rec *,FILE *);

#endif
```

```
/*=====================================================*
 *                ALPRO.C                              *
 * The following functions load in a .pr0 file from    *
 * probench, the data read is then formed into a tree  *
 * as used by the test diagrammer. Full details are    *
 * contained below                                     *
 *=====================================================*/

#include "alpro.h"

/*=====================================================*
 * load_profile takes a filename and tree as arguments *
 * the file is opened, then the header processed to    *
 * check that it is a valid probench file.             *
 *                                                     *
 * The root box is then read, the boxes that follow    *
 * are added onto the tree in different ways.          *
 *                                                     *
 * The goback number is used to find the parent box,   *
 * if the current box is the first child then it is    *
 * created under the parent, but if it is not then it  *
 * is amended onto the list of children                *
 *=====================================================*/
void load_profile(char *filename,TREE *tree)
{
    BOX *boxp,*boxp2;
    int current_box_no;
    struct header_rec diag_header;
    BOXLIST *boxlp;
    FILE *infile=open_input(filename);

    if(!process_header(filename,&diag_header,infile)) /* check header */
       exit();

    boxp=tree->root;            /* point to the root box */

    if(!read_box(infile,boxp))  /* load in the details of the root box */
       exit();
```

```
    current_box_no=2;
    boxp=create_box();          /* create the next box */


    while(read_box(infile,boxp)) /* while there is still a box to read */
       {
        boxp->box_number=current_box_no++; /* assign the box a number,
                                              and increment the number */

        /* find parent ie. the box with
           number, current - goback    */

        boxp2=find_box(boxp->box_number -
                       boxp->probench_box.goback_num,
                       tree);

        if(boxp2->children==NULL) /* this is the parents first child */
          {                       /* so add directly under */
           add_under(boxp2,boxp,tree);
           tree->total_boxes++;        /* increment total boxes */
          }
        else{                          /* add after last child */
             boxlp=boxp2->children;    /* get first child */

             while(boxlp->link!=NULL)
                boxlp=boxlp->link;     /* find last child */

             boxp2=boxlp->box;         /* get last child then */
                                       /* add to the right of */
                                       /* this child */
             add_right(boxp2,boxp,tree);
                 tree->total_boxes++;/* increment total boxes */

                }
       boxp=create_box();   /* create next box */
     }
   free(boxp);          /* delete spuriously created box */
      close_file(infile); /* close the input file */
}
```

```
/*==================================================*
 * Read box reads in the box details from the file,  *
 * It skips any comments and converts the ops arrays *
 * into a linked list as used by the diagrammer      *
 *==================================================*/
int read_box(FILE *infile, BOX *boxp)
{
   short loop,loop2,ops_no;
   char g,als_op_num[4];
   BOX_OPS boxes_ops;
   OPLIST *opl,*lastopl;

   /*read the box, return if it failed */
   if(fread((void *)&(boxp->probench_box),sizeof(BOX_REC),1,infile)!=1)
               return(0);

   /* check that i have read a box */
   if(boxp->probench_box.rec_type != BOX_RECORD)
        return(0); /* oops read the trailer */

   /* skip comment */
   for(loop=0;loop<boxp->probench_box.comment_size;loop++)
         g=getc(infile);



   if(boxp->probench_box.op_indic) /* if the box has ops */
     {
      /* read the first set */
      if(fread(&boxes_ops,sizeof(BOX_OPS),1,infile)!=1)
                  fatal_error("error on read ops");

      ops_no=boxes_ops.op_count; /* find out how many there are */

      opl=create_op();                    /* create first list */
      opl->op_type=boxes_ops.op_type[0]; /* set type */

      /* the op numbers on file are stored as 3 characters
```

```
        we use an integer so prepare to convert it */

         for(loop2=0;loop2<3;loop2++)
                 als_op_num[loop2]=boxes_ops.op_num[0][loop2];
         als_op_num[3]='\0';

         opl->op_num=str_to_int(als_op_num); /* convert first op num */

         boxp->op=opl;        /* get ready to do the rest */
         lastopl=opl;         /* set the last op to this one */
         opl=opl->next_op;

      for(loop=1;loop<OP_GRP_SIZE;loop++) /* loop through array */
        {
         if(loop<ops_no) /* if we still have some to process */
            {
              opl=create_op();            /* create op */
              lastopl->next_op=opl;       /* connect this and
                                            the last together*/

              /* convert op_num as before */
              opl->op_type=boxes_ops.op_type[loop];
              for(loop2=0;loop2<3;loop2++)
                   als_op_num[loop2]=boxes_ops.op_num[loop][loop2];
              als_op_num[3]='\0';
              opl->op_num=str_to_int(als_op_num);

              /* get ready to do the next op */
              lastopl=opl;
              opl=opl->next_op;
            }
        }

      ops_no -= OP_GRP_SIZE; /* total is less the number read */

      while(boxes_ops.next_ops!=NULL) /* if there are more */
        {
         if(fread(&boxes_ops,sizeof(BOX_OPS),1,infile)!=1)
            fatal_error("error on read ops");
```

```
        for(loop=0;loop<OP_GRP_SIZE;loop++)
          {
           if(loop<ops_no)
             {
              opl=create_op();
              lastopl->next_op=opl;
              opl->op_type=boxes_ops.op_type[loop];
              for(loop2=0;loop2<3;loop2++)
                  als_op_num[loop2]=boxes_ops.op_num[loop][loop2];
              als_op_num[3]='\0';
              opl->op_num=str_to_int(als_op_num);
              lastopl->next_op=opl;
              opl=opl->next_op;
             }
          }

        ops_no -= OP_GRP_SIZE;
       }
      getc(infile);getc(infile);
     }

   return(1);
}

/*==================================================*
 * process header checks that there is a header  *
 * and that it is a valid probench one           *
 *==================================================*/
int process_header(char *filename,
                   struct header_rec *diag_header,
                   FILE *infile)
{

   if(fread((void *)diag_header,sizeof(struct header_rec),1,infile)!=1)
       {
        printf("file %s is not a valid profile \n",filename);
        return(0);
       }
```

```
    if(diag_header->file_check_no!=PRO_CHECK_NO)
        {
         printf("file %s is not a valid profile \n",filename);
         return(0);
        }

    return(1);
}
```

G.4 General Code

The following code was used in a number of places to provide a consistent handling of certain features in the programs.

G.4.1 File Handling

```
/*=================================*
 *              ALFILE.H           *
 * declarations of simplified file *
 * initiation routines             *
 *=================================*/

#ifndef ALFILE
#define ALFILE

#include <stdio.h>

extern FILE *open_input(char *);
extern FILE *open_output(char *);
extern void close_file(FILE *);

#endif
```

```
/*==================================================*
 *                    ALFILE.C                      *
 * Simplified file handling routines to open and    *
 * close files, if a file fails to be opened then *
 * the program is exited with an error message      *
 *==================================================*/

#include "alfile.h"

/*==================================================*
 * open_input file opens infile_name as input if *
 * it can't be opened then an error message is    *
 * printed and the program exits                  *
 *==================================================*/
FILE *open_input(char *infile_name)
     {

     FILE *infile;

     if((infile = fopen(infile_name,"r")) == NULL)
         {
          printf(" cannot open input file %s\n",infile_name);
          exit(1);
         }

     return(infile);
     }

/*==================================================*
 * Open_output file opens outfile_name as output, if *
 * it can't be opened then an error message is       *
 * printed and the program exits                     *
 *==================================================*/
FILE *open_output(char *outfile_name)
    {
      FILE *outfile;

      if((outfile = fopen(outfile_name,"w")) == NULL)
            {
```

```
            printf(" cannot open output file %s\n",outfile_name);
            exit(1);
           }

      return(outfile);
    }

/*=====================================*
 * close_file closes the supplied file *
 * function  supplied to be consistent *
 * with above calls                    *
 *=====================================*/
void close_file(FILE *closeme)
    {
      fclose(closeme);
    }
```

G.4.2 Error Handling

```
/*================================================*
 *                  ERRORS.H                      *
 * declarations of the error displaying functions *
 *================================================*/
#ifndef ERRORS
#define ERRORS

#include <stdio.h>

extern void fatal_error(char *);
extern void error(char *);


#endif
```

```
/*==============================================================*
 *                ERRORS.C                                      *
 * the error functions used in the programs,                    *
 * fatal error exits the program after displaying the error *
 * error carries on                                             *
 *==============================================================*/


#include "errors.h"

void fatal_error(char *error_mess)
{
        printf("FATAL ERROR: %s",error_mess);
        exit(1);
}

void error(char *error_mess)
{
        printf("ERROR: %s",error_mess);
```

```
}
```

G.4.3 String Conversion

```
/*=====================================================*
 *              ALUTIL.H                               *
 *  Any utilities that are needed in the project       *
 *  are not relevant in any other part of the project *
 *  and are not supplied by the compiler               *
 *=====================================================*/
#ifndef ALUTIL
#define ALUTIL

/*=====================================================*
 * str_to_int turns the supplied character string into *
 * the returned integer value                          *
 *=====================================================*/
static int str_to_int(char *con)
   {
     int count=0;

     while(*con!='\0')
        {
         count=count*10;
         count+=(*con)-'0';
         con++;
        }
     return(count);
   }

#endif
```

G.5 Redundant Code

The follwoing code provided a means of creating strings dynamically but it soon became redundant as the programs were written using fixed length arrays, and when strings were written to a file preceded by their length.

```
/* ALSTRING.h a collection of functions to manipulate dynamic strings */
#ifndef ALSTRING
#define ALSTRING

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


/* The following define is the increment for string resizing */
#define INCR 10

/* the new structure for dynamic strings */
typedef struct {
                int max_size;           /*size allocated for string*/
                int curr_size;          /*current size */
                char *string_part;      /*pointer to string*/
               }DYNAMIC_STRING;        /*new type allows resizing etc*/

/* Function definitions*/
extern DYNAMIC_STRING *create_dynamic_string(void);
extern void append_to_string(char,DYNAMIC_STRING *);
extern void resize_by(DYNAMIC_STRING *,int);
extern void reset_string(DYNAMIC_STRING *);
extern void force_upper(char *);
extern void destroy_dynamic(DYNAMIC_STRING *);
extern char *extract_then(DYNAMIC_STRING *);
#endif
```

```
#include "alstring.h"

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



/**********
***********              FUNCTIONS
***********/
/*      function allocates memory for a dynamic string, handles out
        out of memory errors, initialises the string and returns its
        address*/

DYNAMIC_STRING *create_dynamic_string()
  {
    DYNAMIC_STRING *dst;
    char *st;

    if((dst=(DYNAMIC_STRING *)malloc(sizeof(DYNAMIC_STRING)))==NULL)
            fatal_error("no more room for dynamic string");

     /* allocates room for 1 char !! */
    if((st=(char *)malloc(sizeof(char)))==NULL)
            fatal_error("no more room for string part");


     /* make the string end legally */
     st[0]='\0';

     dst->max_size=sizeof(char);
     dst->curr_size=0;
     dst->string_part=st;

     return dst;
  }

/*      append a character to a dynamic string, resize if necessary */
void append_to_string(char ch,DYNAMIC_STRING *dynamic_string)
```

```
   {
     /* if the next slot is the last slot then resize */
     if((dynamic_string->curr_size+1)==dynamic_string->max_size)
            resize_by(dynamic_string,INCR);

     /* curr_size acts as a subscript to the last character */
     dynamic_string->string_part[dynamic_string->curr_size]=ch;

     dynamic_string->curr_size++;
     dynamic_string->max_size++;
     dynamic_string->string_part[dynamic_string->curr_size]='\0';
   }

/*      resize a dynamic string by allocating memory, copying from
        the old memory locations to the new one, and freeing the
        old locations */
void resize_by(DYNAMIC_STRING *dynamic_string, int incr)
    {
      char *increased;

      if((increased=(char *)realloc((char *)dynamic_string->string_part,
                                    dynamic_string->max_size+incr))==NULL)
             fatal_error("no room to resize a dynamic string");


      strcpy(increased,dynamic_string->string_part);
      free(dynamic_string->string_part);

      dynamic_string->string_part=increased;
      dynamic_string->max_size+=incr;
    }

/* resets a dynamic string to 1 char length */
void reset_string(DYNAMIC_STRING *dst)
   {
     char *st;

     free(dst->string_part);
```

```
    /* allocates room for 1 char !! */
    if((st=(char *)malloc(sizeof(char)))==NULL)
         fatal_error("no more room for string part");


    /* make the string end legally */
    st[0]='\0';

    dst->max_size=sizeof(char);
    dst->curr_size=0;
    dst->string_part=st;

  }

/* force uppercase */
void force_upper(char *string)
   {
     while(*string!='\0')
       {
        *string=toupper(*string);
        string++;
       }
   }

/* totally destroy a dynamic string */
void destroy_dynamic(DYNAMIC_STRING *dst)
   {
     free(dst->string_part);
     free(dst);
   }

/* return the string but destroy the shell */
char *extract_then(DYNAMIC_STRING *dst)
   {
     free(dst);
     return(dst->string_part);
   }
```

G.6 Trie Input Data

This file contains all the COBOL keywords, in the format required by the Trie input routine, *'s are used to tell the Trie code that a unique number should be given to this keyword and its associated synonyms.

```
*ACCEPT
*ACCESS
*ADD
*ADVANCING
*AFTER
*ALL
*ALPHABET
*ALPHABETIC
*ALPHABETIC-LOWER
*ALPHABETIC-UPPER
*ALPHANUMERIC
*ALPHANUMERIC-EDITED
*ALSO
*ALTER
*ALTERNATE
*AND
*ANY
*ARE
*AREA
AREAS
*ASCENDING
*ASSIGN
*AT
*AUTHOR
*BEFORE
*BINARY
*BLANK
*BLOCK
*BOTTOM
*BY
*CALL
```

```
*CANCEL
*CD
*CF
*CH
*CHARACTER
*CHARACTERS
*CLASS
*CLOSE
*CODE
*CODE-SET
*COLLATING
*COLUMN
*COMMA
*COMMON
*COMMUNICATION
*COMP
COMPUTATIONAL
*COMPUTE
*CONFIGURATION
*CONTAINS
*CONTENT
*CONTINUE
*CONTROL
*CONTROLS
*CONVERTING
*COPY
*CORR
CORRESPONDING
*COUNT
*CURRENCY
*DATA
*DATE
*DATE-COMPILED
*DATE-WRITTEN
*DAY
*DAY-OF-WEEK
*DE
*DEBUG-CONTENTS
*DEBUG-ITEM
```

```
*DEBUG-LINE
*DEBUG-NAME
*DEBUG-SUB-1
*DEBUG-SUB-2
*DEBUG-SUB-3
*DEBUGGING
*DECIMAL-POINT
*DECLARATIVES
*DELETE
*DELIMITED
*DELIMITER
*DEPENDING
*DESCENDING
*DESTINATION
*DETAIL
*DISABLE
*DISPLAY
*DIVIDE
*DIVISION
*DOWN
*DUPLICATES
*DYNAMIC
*EGI
*ELSE
*EMI
*ENABLE
*END
*END-ADD
*END-CALL
*END-COMPUTE
*END-DELETE
*END-DIVIDE
*END-EVALUATE
*END-IF
*END-MULTIPLY
*END-OF-PAGE
*END-PERFORM
*END-READ
*END-RECEIVE
```

```
*END-RETURN
*END-REWRITE
*END-SEARCH
*END-START
*END-STRING
*END-SUBTRACT
*END-UNSTRING
*END-WRITE
*ENVIRONMENT
*EOP
*EQUAL
*ERROR
*ESI
*EVALUATE
*EXCEPTION
*EXIT
*EXTEND
*EXTERNAL
*FALSE
*FD
*FILE
*FILE-CONTROL
*FILLER
*FINAL
*FIRST
*FOOTING
*FOR
*FROM
*GENERATE
*GIVING
*GLOBAL
*GO
*GREATER
*GROUP
*HEADING
*HIGH-VALUE
HIGH-VALUES
*I-O
*I-O-CONTROL
```

```
*IDENTIFICATION
*IF
*IN
*INDEX
*INDEXED
*INDICATE
*INITIAL
*INITIALIZE
*INITIATE
*INPUT
*INPUT-OUTPUT
*INSPECT
*INSTALLATION
*INTO
*INVALID
*IS
*JUST
*JUSTIFIED
*KEY
*LABEL
*LAST
*LEADING
*LEFT
*LENGTH
*LESS
*LIMIT
*LIMITS
*LINAGE
*LINAGE-COUNTER
*LINE
*LINE-COUNTER
*LINES
*LINKAGE
*LOCK
*LOW-VALUE
LOW-VALUES
*MERGE
*MESSAGE
*MODE
```

```
*MOVE
*MULTIPLE
*MULTIPLY
*NATIVE
*NEGATIVE
*NEXT
*NO
*NOT
*NUMBER
*NUMERIC
*NUMERIC-EDITED
*OBJECT-COMPUTER
*OCCURS
*OF
*OFF
*OMITTED
*ON
*OPEN
*OPTIONAL
*OR
*ORDER
*ORGANIZATION
*OTHER
*OUTPUT
*OVERFLOW
*PACKED-DECIMAL
*PADDING
*PAGE
*PAGE-COUNTER
*PERFORM
*PF
*PH
*PIC
PICTURE
*PLUS
*POINTER
*POSITION
*POSITIVE
*PRINTING
```

```
*PROCEDURE
*PROCEDURES
*PROCEED
*PROGRAM
*PROGRAM-ID
*PURGE
*QUEUE
*QUOTE
QUOTES
*RANDOM
*RD
*READ
*RECEIVE
*RECORDS
*REDEFINES
*REEL
*REFERENCES
*RELATIVE
*RELEASE
*REMAINDER
*REMOVAL
*RENAMES
*REPLACE
*REPLACING
*REPORT
*REPORTING
*REPORTS
*RESERVE
*RESET
*RETURN
*REWIND
*REWRITE
*RF
*RH
*RIGHT
*ROUNDED
*RUN
*SAME
*SD
```

```
*SEARCH
*SECTION
*SECURITY
*SEGMENT
*SEGMENT-LIMIT
*SELECT
*SEND
*SENTENCE
*SEPARATE
*SEQUENCE
*SEQUENTIAL
*SET
*SIGN
*SIZE
*SORT
*SORT-MERGE
*SOURCE
*SOURCE-COMPUTER
*SPACE
SPACES
*SPECIAL-NAMES
*STANDARD
*STANDARD-1
*STANDARD-2
*START
*STATUS
*STOP
*STRING
*SUB-QUEUE-1
*SUB-QUEUE-2
*SUB-QUEUE-3
*SUBTRACT
*SUM
*SUPPRESS
*SYMBOLIC
*SYNC
SYNCHRONIZED
*TABLE
*TALLYING
```

```
*TAPE
*TERMINAL
*TERMINATE
*TEST
*TEXT
*THAN
*THEN
*THROUGH
THRU
*TIME
*TIMES
*TO
*TOP
*TRAILING
*TRUE
*TYPE
*UNIT
*UNSTRING
*UNTIL
*UP
*UPON
*USAGE
*USE
*USING
*VALUE
VALUES
*VARYING
*WHEN
*WITH
*WORKING-STORAGE
*WRITE
*ZERO
ZEROES
ZEROS
```

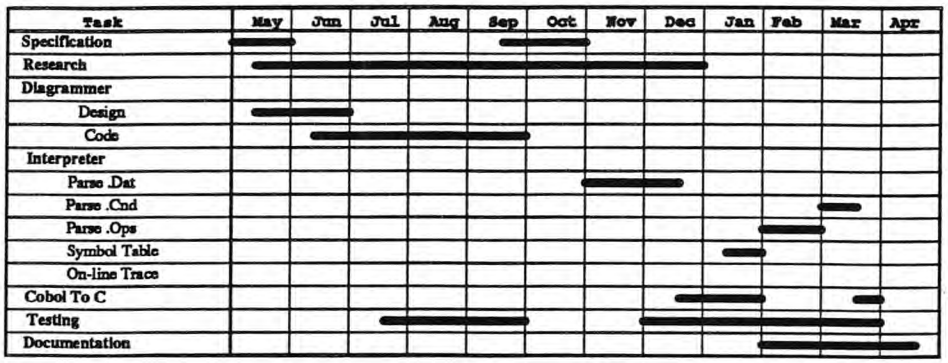

Figure 18: Project Gantt Chart

H Project Management Information

H.1 Gantt Chart

H.2 Project Diary